

Docker for developers

DevOps Course

JOHN BRYCE
Leading in IT Education
matrix company





Yaniv Cohen

PortFolio

<https://goo.gl/9ivmRP>

Linkedin

<https://www.linkedin.com/in/yanivos/>

EMAIL

yanivomc@gmail.com



Clone Git

`git clone https://github.com/yanivomc/seminars.git`

- By the end of this session
 - You'll be familiar with Docker Concepts & Base Commands
 - Configure Dockers using DockerFile And Passing Properties To It
 - Run Standalone Jar in docker
 - Operate Docker Hub (Push)
 - Build Docker Image with Maven
 - Advance Docker - Network and Docker compose
 - Utilize docker compose for your own CD in your local development env

Questions for you...

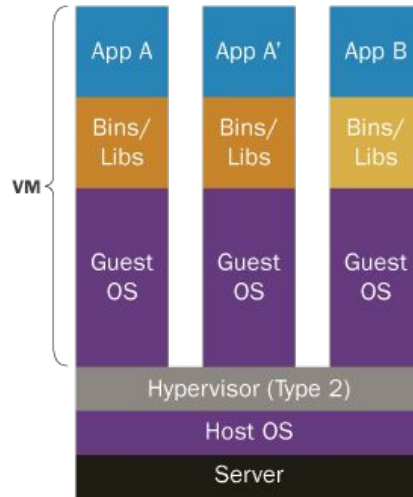
- What Do You Know About Docker?
- Who Used Docker For Development / QA / STG / PROD?
- Who Tried & Failed Implementing Docker

What is Docker

Docker is an open platform for developing, shipping, and running applications.

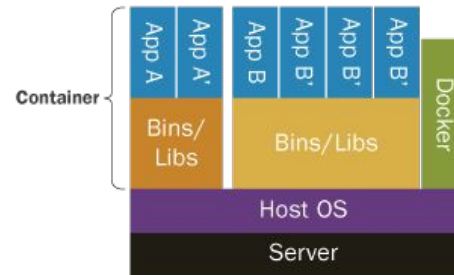
Docker allows you to package an application with all of its dependencies into a standardized unit for software development.

Containers VS. VMs



Virtual Machines

Containers are isolated, but share OS and, where appropriate, bins/libraries



Containers

Docker Benefits Upon VMs

- Small to tiny images - Few hundred MB's for OS + Application (5MB for full OS - [Alpine](#)) VS. Gigabytes in VM's
- Very small footprint on the host machine (CPU, RAM Impact) as Docker only use what it required instead of building a complete Operating system per VM.
- Containers use up only as many system resources as they need at a given time. VMs usually require some resources to be permanently allocated before the virtual machine starts.
- Direct hardware access. Applications running inside virtual machines generally cannot access hardware like graphics cards on the host in order to speed processing.
Containers Can (ex. [Nvidia](#))
- Microservice in nature and integrations (API's) for whatever task required.
- Portable, Fast (Deployments , Migration , Restarts and Rollbacks) and Secure
- Can run anywhere and everywhere
- Simplify DevOps
- Version controlled
- Open Source

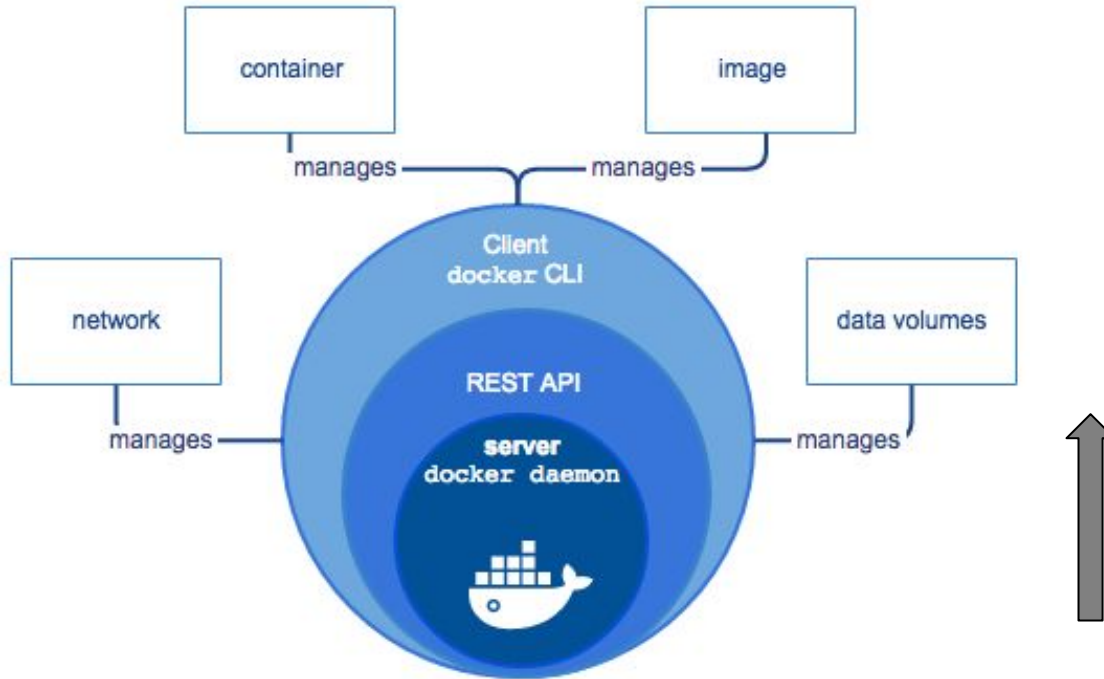
Common Use Cases for Docker

- CI / CD
- Fast Scaling application layers for overcoming application performance limitations.
- For Sandboxed environments (Development, Testing , Debugging)
- Local development environment (no more “ It ran on my laptop...”)
- Infrastructure as a CODE made easy with docker
- Multi-Tier applications (Front End , Mid Tier (Biz Logic) , Data Tier) / Microservices
- Building PaaS , Saas

- Architecture: Linux X86-64
- Written in: GoLang (On March 13, 2014, with the release of version 0.9, Docker dropped LXC as the default execution environment which is an operating system level virtualization and replaced it with its own libcontainer library written in the Go programming language)
- Engine: Client - Server (Daemon) Architecture
- Namespace: Isolation of process in linux where one process cant “See” the other process
- Control Groups: Linux Kernel capability to limit and isolate the resource usage (CPU, RAM, disk I/O, network etc..) of a collection of process
- Container format: libcontainer - Go implementation for creating containers with namespaces, control groups and File system capabilities access control

Docker Architecture

Overview



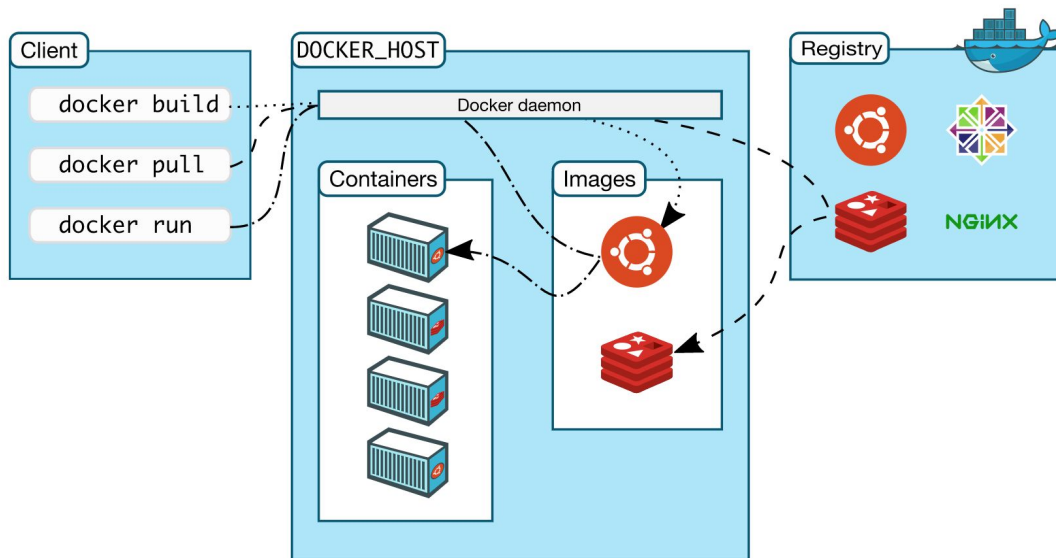
What is docker - Technical Aspect

Docker Architecture

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing your Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

Docker Architecture

Docker Architecture



Docker Components

- Engine
- Daemon
- (Docker) Client
- Docker Registries
- Docker Objects
- Machine
- Compose
- Swarm

Docker Components

Engine

- A server which is a type of long-running program called a daemon process (the dockerd command).
- A REST API which specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
- A command line interface (CLI) client (the docker command).

Docker Components

Daemon

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

Docker Client

- The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

Docker Registries

- A Docker registry stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.
- When one use “docker pull / push / run” commands, the required images are pulled from the configured registry.

Docker Objects

- Images
 - a. Read Only template with instruction for creating a Docker Container. Often, an Image is based on another image with some additional customization.
 - b. Self own images that are fully created by you using DockerFile with a simple syntax where every instruction control a different Layer in the image. Once a change is made to a specific layer, a rebuild of the image will change only the updated layers. This what makes images small, fast and lightweight in compared to other virtualization solutions

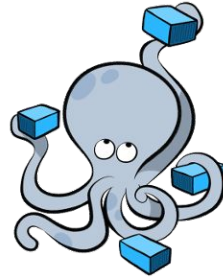
Docker Objects

- Containers
 - a. A container is a runnable instance of an image. You can create, start, stop, move, or delete a container using the Docker API or CLI. You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
 - b. Container is defined by its image as well as any configuration options we provide to it when created or when we start it

Docker Objects

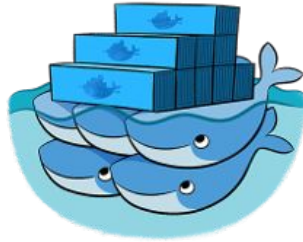
- Services
 - a. Allow you to scale containers across multiple Docker daemons, which all work together as a **swarm** with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API. A service allows you to define the desired state, such as the number of replicas of the service that must be available at any given time. By default, the service is load-balanced across all worker nodes. To the consumer, the Docker service appears to be a single application. Docker Engine supports swarm mode in Docker 1.12 and higher.

Docker Compose



A tool for defining and running complex applications with
Docker (eg multi-container application ex. LAMP)
With a single file

Docker Swarm



A Native Clustering tool for Docker. Swarm pools together several Docker hosts and exposed them as a single virtual Docker host. It scale up to multiple hosts

Docker Components

Good to know:

Docker Machine



A Tool which makes it easy to create Docker Hosts on Operating systems that does not support docker natively, or on cloud providers and inside your datacenter.



INSTALLING DOCKER

Windows 10 Enterprise / Educational



[DOWNLOAD HERE](https://docs.docker.com/docker-for-windows/)

<https://docs.docker.com/docker-for-windows/>

Windows 10 Enterprise / Educational

1. Turn windows features on or off
 - a. Enable HYPER V
 - b. Restart

Windows 10 Check Functionality

1. Open a shell (`cmd.exe` , PowerShell, or other).
2. Run some Docker commands, such as `docker ps` , `docker version` , and `docker info` .

Here is the output of `docker ps` run in a powershell. (In this example, no containers are running yet.)

```
PS C:\Users\jdoe> docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
--------------	-------	---------	---------	--------	-------

Here is an example of command output for `docker version` .

```
PS C:\Users\Docker> docker version
Client:
Version:      17.03.0-ce
API version:  1.26
Go version:   go1.7.5
Git commit:   60ccb22
Built:        Thu Feb 23 10:40:59 2017
OS/Arch:      windows/amd64

Server:
Version:      17.03.0-ce
API version:  1.26 (minimum version 1.12)
Go version:   go1.7.5
Git commit:   3a232c8
Built:        Tue Feb 28 07:52:04 2017
OS/Arch:      linux/amd64
Experimental: true
```



Let's Start

```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest sh
```

- 'docker run' will run the container
- This will not restart an already running container, just create a new one
- docker run [options] IMAGE [command] [arguments]
 - a. [options] modify the docker process for this container
 - b. IMAGE is the image to use
 - c. [command] is the command to run inside the container (entry point to hold the container running)
 - d. [arguments] are arguments for the command

```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest sh
```

- 'docker run' will run the container
 - a. -i - Interactive mode
 - b. -t - Allocate pseudo TTY - or not Terminal will be available
 - c. -d - Run in the background (Daemon style)
 - d. --name - Give the container a name or let Docker to name it
 - e. -p [local port] : [container port] - Forward local port to the container port

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
98debfed4458	alpine:latest	"sh"	Less than a second ago	Up 1 second	0.0.0.0:8080->80/tcp	dockerlearning


```
docker run -i -t -d --name dockerlearning -p 8080:80 alpine:latest sh
```

- Pulls the alpine:latest image from the registry (if not existed on our station)
 - a. Run “docker images” to see what images already downloaded / in use locally
- Creates new container
- Allocate FS and Mounts a read-write Layer
- Allocates network/bridge interface
- Set up an IP Address
- Executes a process that we specify (in this scenario - “sh” as alpine release doesn't have bash)
- Captures and provides application outputs

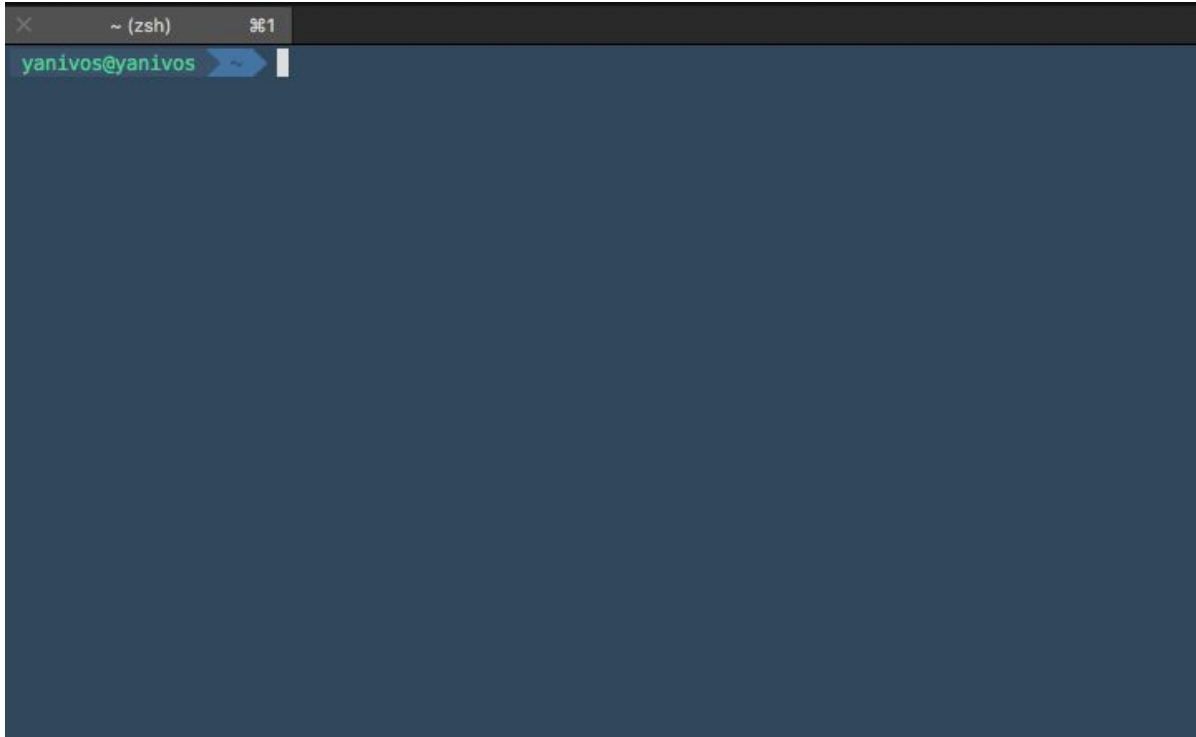
Docker Examples

- Pull / Run an image
- SSH into a container
- View Logs
- Docker Volume
- Using Dockerfile - Building our own Jar
- Package an app and push it to a repo

Common Docker Commands

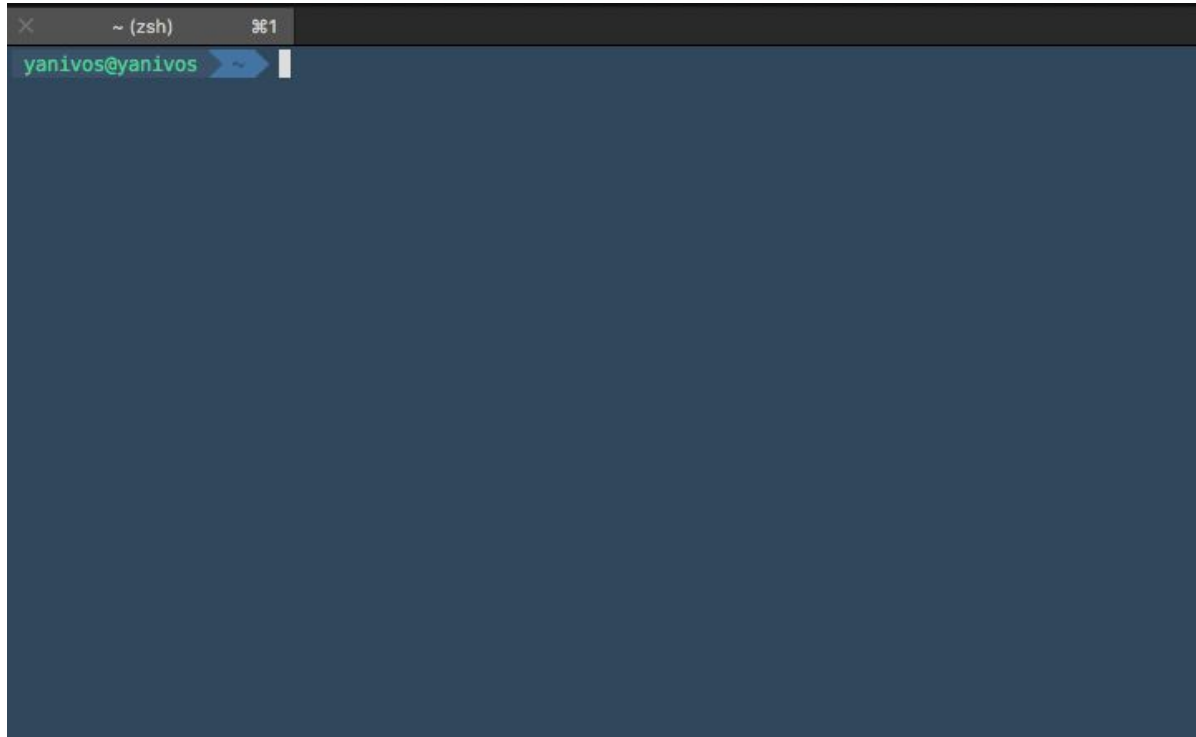
```
// General info
man docker // man docker-run
docker help // docker help run
docker info
docker version
docker network ls
// Images
docker images // docker [IMAGE_NAME]
docker pull [IMAGE] // docker push [IMAGE]
// Containers
docker run
docker ps // docker ps -a, docker ps -l
docker stop/start/restart [CONTAINER]
docker stats [CONTAINER]
docker top [CONTAINER]
docker port [CONTAINER]
docker inspect [CONTAINER]
docker inspect -f "{{ .State.StartedAt }}" [CONTAINER]
docker rm [CONTAINER]
```

Running simple shell



A terminal window with a dark blue background. The title bar at the top shows a close button, the text "~ (zsh)", and a window icon. The prompt "yanivos@yanivos" is displayed in green text, followed by a blue arrow icon and a white cursor. The rest of the terminal area is empty.

Building & Running Mysql On docker



Why not to run SSH inside a container

- We can...
- Docker is designed for one command per container - Now we run two
- If any update or modification is needed, We need to change our setup and not the docker image..
- If you still want to review something... SSH it.



Docker Advanced

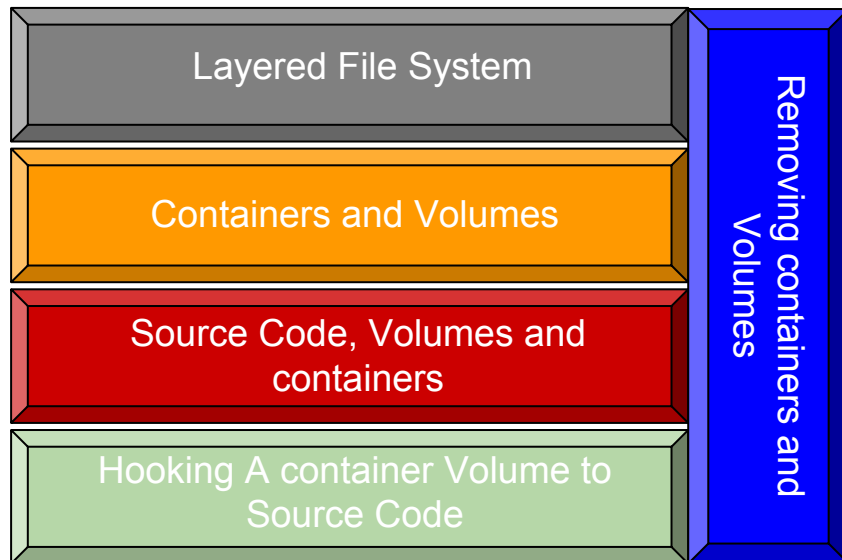
- Volumes - Hooking Source code into a container
- Networking and communications
- Building Custom Images with DockerFile
- Building Custom images with Docker Compose (v3 YAML)
- Working with images
- Building a Microservice Project
- Working with Private Registries



HOOKING SOURCE CODE

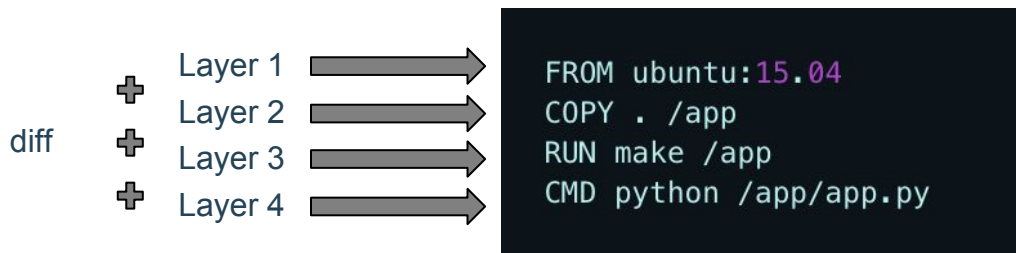
Module Agenda

To understand how we can hook our source code into a container,
We will go over the following:



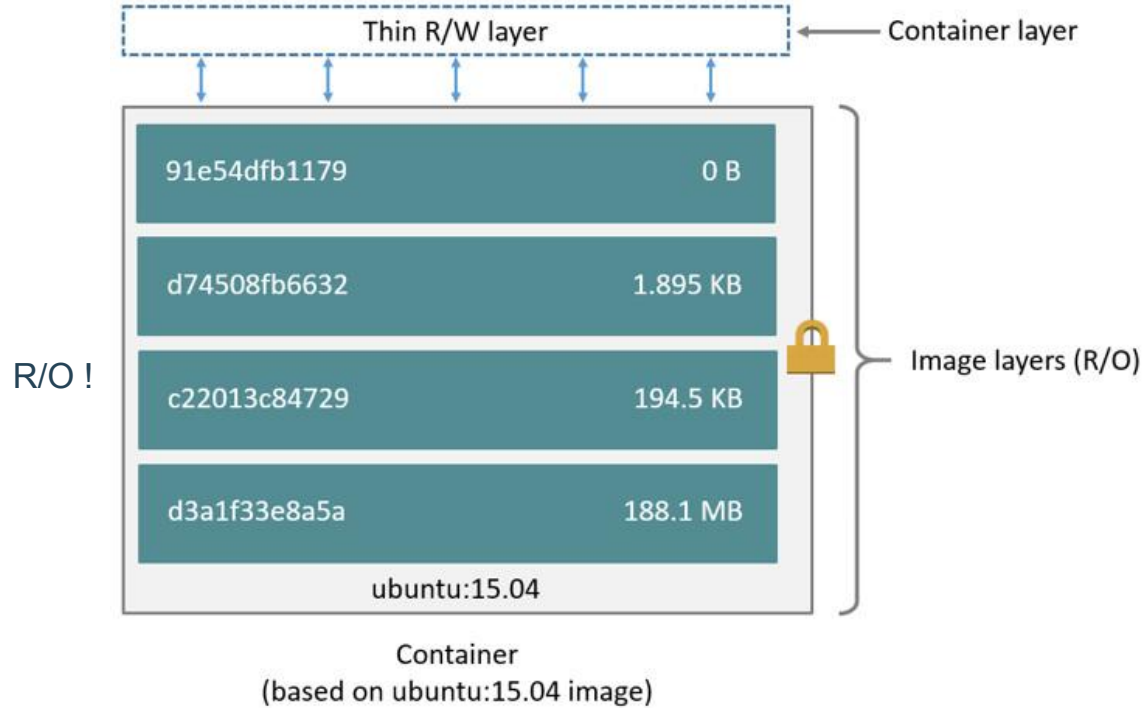
- Images and Layers

A Docker image is built up from a series of layers. Each layer represents an instruction in the image's Dockerfile. Each layer except the very last one is read-only. Consider the following Dockerfile



Each layer is only a set of differences from the layer before it. The layers are stacked on top of each other. When you create a new container, you add a new writable layer on top of the underlying layers. This layer is often called the “**container layer**”. All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer.

Layered FS

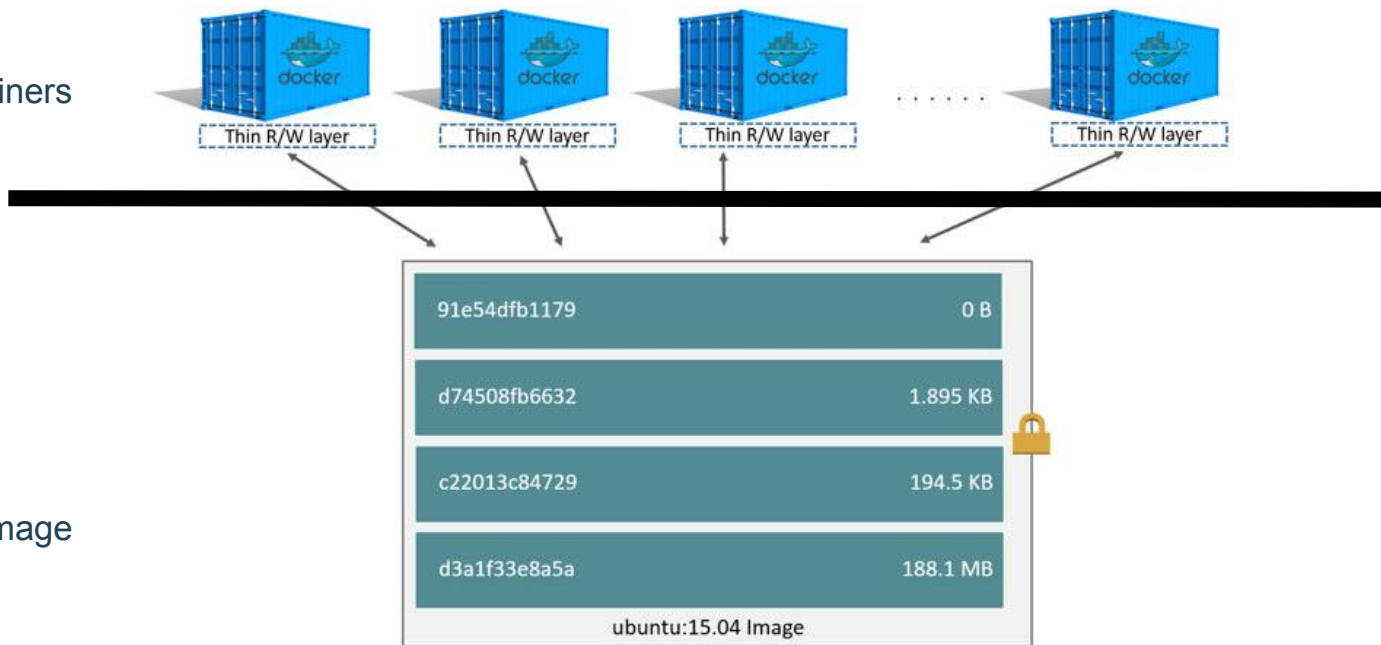


- **Containers and Layers**

The major difference between a container and an image is the top writable layer. All writes to the container that add new or modify existing data are stored in this writable layer. When the container is deleted, the writable layer is also deleted. The underlying image remains unchanged. Because each container has its own writable container layer, and all changes are stored in this container layer, multiple containers can share access to the same underlying image and yet have their own data state. The diagram below shows multiple containers sharing the same Ubuntu 15.04 image.

Layered FS

Different Containers



Using same Image

Note: If we need multiple images to have shared access to the exact same data, we store this data in a Docker **volume** and mount it into your containers.



SO HOW DO WE GET OUR SOURCE CODE INTO A CONTAINER?



Containers and Volumes

- What is a Volume

Special type of directory in a container typically referred to as a “data volume”

- Can be shared and reused among one or many containers
- Updates to an image won't affect a data volume
- Data volumes are persisted even after container deletion
- Volumes are OS agnostic. They can run on Linux and windows containers
- Volumes drivers allow us to store volumes on remote hosts or cloud providers.
- Volumes can be encrypted or to add other functionality
- A new volume content can be pre-populated by a container



Containers and Volumes

Follow through

- **Create and manage volumes:**

What will we achieve in the following follow through session:

- Creating new volume
 - Inspecting
 - Removing
- Start a container[s] with a volume

Follow through

RUN

```
docker run -dti --name alpine1 --mount target=/app alpine ash
```

INSPECT

```
docker inspect alpine1
```

```
},  
"Mounts": [  
  {  
    "Type": "volume",  
    "Name": "e2c79e12b3f8b90888688da603cca4c2297ee96f2b914a601378e5d944f17214",  
    "Source": "/var/lib/docker/volumes/e2c79e12b3f8b90888688da603cca4c2297ee96f2b914a601378e5d944f17214/_data",  
    "Destination": "/app",  
    "Driver": "local",  
    "Mode": "z",  
    "RW": true,  
    "Propagation": ""  
  }  
],
```

STOP AND DELETE CONTAINER

```
docker stop alpine1 && docker rm alpine1
```

Follow through 2

Creating a VOLUME managed by docker FS and share it with multiple containers

RUN

```
docker volume create fs_shared
```

LIST VOLUMES

```
docker volumes ls
```

```
local          fs_shared
```

RUN AND MOUNT

```
docker run --rm -tdi --name alpine1 --mount source=fs_shared,target=/app alpine ash
```

```
docker run --rm -tdi --name alpine2 --mount source=fs_shared,target=/app alpine ash
```

```
docker run --rm -tdi --name alpine3 --mount source=fs_shared,target=/app alpine ash
```

LAB

Attach to running containers, create files and verify files gets updated on all containers

Disconnect sequence `Ctrl + p + Ctrl + q`



Containers and Volumes

BIND MOUNTS

- Bind Mounts

Bind mounts have been around since the early days of Docker. Bind mounts have limited functionality compared to volumes. When you use a bind mount, a file or directory on the host machine is mounted into a container. The file or directory is referenced by its full or relative path on the host machine. By contrast, when you use a volume, a new directory is created within Docker's storage directory on the host machine, and Docker manages that directory's contents.

COMMAND EXAMPLE

-v

```
docker run --rm -tdi -v "$(pwd)/source:/app [image] [CMD]
```

--mount

```
docker run --rm -tdi --mount type=bind,source="$(pwd)/source,target=/app [image] [CMD]
```

- **BIND MOUNTS USING -V OR --MOUNT ?**

- Both will provide the same outcome but as -v /--volume exists since day 1 in docker and --mount was introduced since docker 17.06 it became normal and easier to use --mount.



Containers and Volumes

LAB: BIND MOUNTS

- **Create and manage bind mount:**
 - Create new host local project folder called “jb_docker” and cd into it
 - Create 2 alpine nodes and share new local folder called source1 using --mount
 - Create 2 alpine nodes and share new local folder called source2 using -v
 - What happened when you tried creating a shared host folder with --mount without first creating the folder manually ? and what happened when you were using -v
 - Inspect the new volumes and containers
 - Validate shared folder by creating files and make sure the exists on both containers
 - Stop all docker containers and Make sure containers got deleted



Containers and Volumes

LAB: Running BootStrap app in a container

- **Hook SpringBoot Jar into a container:**

- Cd into your “jb_docker” folder
 - Copy from your cloned git the demo artifact to ./source
seminars/docker/dockercompose/artifacts/**spring-boot-rest-example-0.4.0.war**
- Run 1 new container
 - Name: web_api
 - Mount Using -v or --mount
 - source: ./source
 - Target: /app
 - Image: **frolvlad/alpine-oraclejdk8:slim**
 - CMD: **java -jar -Dspring.profiles.active=test /app/spring-boot-rest-example-0.4.0.war**

- **Validate your work:**
 - Run `docker ps` and expect to see the following

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
c9ca53789a18	frolvlad/alpine-oraclejdk8:slim	"java -jar -Dsprin..."	About a minute ago	Up 2 minutes	0.0.0.0:8080->8080/tcp, 0.0.0.0:8091->8091/tcp	web_api

- Run `docker logs` OR `attach` and expect seeing the following (remember `ctrl+p+ctrl+q` to disconnect)

```
2018-03-01 22:11:36.151 INFO 1 --- [main] s.w.s.m.a.RequestMappingHandlerMapping : Mapped "[/error]" onto public java.util.Map<java.lang.String, java.lang.Object> or
int.invoke())
2018-03-01 22:11:36.164 INFO 1 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web.se
2018-03-01 22:11:36.164 INFO 1 --- [main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet.re
2018-03-01 22:11:36.199 INFO 1 --- [main] s.w.s.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.AnnotationC
Mar 01 22:11:34 GMT 2018]; parent: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@4aa8f0b4
2018-03-01 22:11:36.645 INFO 1 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8091 (http)
2018-03-01 22:11:36.683 INFO 1 --- [main] o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 2147483647
2018-03-01 22:11:36.685 INFO 1 --- [main] d.s.w.p.DocumentationPluginsBootstrapper : Context refreshed
2018-03-01 22:11:37.026 INFO 1 --- [main] d.s.w.p.DocumentationPluginsBootstrapper : Found 1 custom documentation plugin(s)
2018-03-01 22:11:37.140 INFO 1 --- [main] s.d.s.w.s.ApiListingReferenceScanner : Scanning for api listing references
2018-03-01 22:11:38.954 INFO 1 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8090 (http)
2018-03-01 22:11:38.985 INFO 1 --- [main] com.khoubyari.example.Application : Started Application in 61.65 seconds (JVM running for 64.564)
2018-03-01 22:11:39.501 INFO 1 --- [nio-8091-exec-1] o.a.c.c.C.[Tomcat-1].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
2018-03-01 22:11:39.502 INFO 1 --- [nio-8091-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started
2018-03-01 22:11:40.484 INFO 1 --- [nio-8091-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 982 ms
```

Try Browsing from your host browser: <http://localhost:809/health>

Did it worked?

- What do you need to do to forward request to port 8080 and 8091 to your docker web_api ?



FINAL SOLUTION

HOOKING YOUR OWN SOURCE CODE

Volumes

```
docker run -tdi --name web_api -v "$(pwd)"/source:/app -p 8080:8080 -p 8091:8091 frolov/alpine-oraclejdk8:slim  
java -jar -Dspring.profiles.active=test /app/spring-boot-rest-example-0.4.0.war
```




STOP AND REMOVE CLEAN UP



NETWORKING

Intro

One of the reasons Docker containers and services are so powerful is that you can connect them together, or connect them to non-Docker workloads. Docker containers and services do not even need to be aware that they are deployed on Docker, or whether their peers are also Docker workloads or not. Whether your Docker hosts run Linux, Windows, or a mix of the two, you can use Docker to manage them in a platform-agnostic way.

What are the common network drivers types?

- **Bridge**

The basic and default driver which is used for standalone containers setup that need to communicate.

- **Overlay**

Connect multiple docker daemons together and enable swarm (cluster) services to communicate with each other. This can be used to facilitate communication between swarm and standalone container or between two standalone containers on different docker daemons.

- **macVLAN**

Macvlan network allow us to assign a MAC address to a container for making it appear as physical device on our network. Usually to be used with legacy or HW required product that must have a MAC and being directly connected to the physical network to operate.

Network Driver Summary

- **Bridge**

User-defined bridge networks are best when you need multiple containers to communicate on the same Docker host.

- **Overlay**

are best when you need containers running on different Docker hosts to communicate, or when multiple applications work together using swarm services - Works with Swarm only

- **macVLAN**

Macvlan network 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



NETWORK

Common practice for user defined bridge setup
HANDS ON LAB

Network - Basics

Follow through

Default Bridge network

The default bridge network is what Docker setup for us automatically.

It's a great way to start but this is **not suitable for production use**

Follow through

We start by inspecting the current network

```
yanivos@ip-10-0-0-25 ~$ docker network ls
```

NETWORK ID	NAME	DRIVER	SCOPE
544e9ba6b7dd	bridge	bridge	local
09b1365bef97	composeelk_esnet	bridge	local
068dea2a1105	downloads_esnet	bridge	local
c2c8e513337c	host	host	local
6f2d144ac291	none	null	local

The default **bridge** network is listed, along with host and none. The latter two are not fully-fledged networks, **but are used to start a container connected directly to the Docker daemon host's networking stack, or to start a container with no network devices**. This follow through will connect two containers to the bridge network

Follow through

Add two new alpine containers with ash as entry point

```
docker run --rm -tdi --name alpine1 alpine ash
docker run --rm -tdi --name alpine2 alpine ash
```

As you recall: The `-tdi` flags means start the container detached (in the background), interactive (with the ability to type into it), and with a TTY (so you can see the input and output). Because we did not specified any `--network` flags, the containers connect to the **default bridge network**

Follow through

Next:

1. Check that the containers are actually running
2. Inspect the network and see what containers are connected to it using
docker network inspect bridge
3. Connect to one of the Alpine containers using **docker attach** and ping the other container with IP and then with its name.

What happened ?

Inspect example

```
yanivos@ip-10-0-0-25 ➤ docker network inspect bridge
[
  {
    "Name": "bridge",
    "Id": "544e9ba6b7dd0829afab0c8599ca78f5dcfa07db93893d730185b2d9ccd9ca4",
    "Created": "2018-02-11T20:10:28.844017437Z",
    "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,
    "ConfigFrom": {
      "Network": ""
    },
    "ConfigOnly": false,
    "Containers": {
      "3032bcbde165cc21fc530e0fbdabb1f8d4923c2eaf713c3d42b42f9054c77115b": {
        "Name": "alpine1",
        "EndpointID": "7809b16709d0ae7a752b5d3e50272d1358347ad51b4ada1d0c49b7bfbef1da4e",
        "MacAddress": "02:42:ac:11:00:02",
        "IPv4Address": "172.17.0.2/16",
        "IPv6Address": ""
      },
      "4caa720e6d2997d0b8e0a4a54f8d69310d47d25cba89515248212e93e4e32e31": {
        "Name": "alpine2",
        "EndpointID": "dd794f05d5f012a2fd169d22e1314335ee79e931838d266def9a9756e50c8688",
        "MacAddress": "02:42:ac:11:00:03",
        "IPv4Address": "172.17.0.3/16",
        "IPv6Address": ""
      }
    }
  }
]
```

Follow through

Pinging containers with IP worked while with name it Failed.

Default Bridge driver does not allow name linking / resolution

Follow through

User Define Bridge network

User define Bridge network provide us with a way to better arrange / build our network topology and communication across containers that connected to the same User Define bridge network along with a DNS Resolution.

Follow through

Creating user Define Bridge network

CREATE NEW BRIDGE NETWORK

```
docker network create dmz
```

INSPECT NETWORK

```
docker network inspect dmz
```

RUN CONTAINER IN DMZ

```
docker run -tdi --rm --name network_test -network dmz alpine ash
```

INSPECT CONTAINER

```
docker inspect network_test
```



NETWORK BRIDGES

LAB

User Define Bridge network

LAB

1. Delete the previous containers (stop and then remove)
2. Create a newly user Defined network bridge named “**alpine-net**” and verify creation with **network ls** and than **Inspect** the network to see that no containers are connected
3. Create 4 new alpine containers with -dit and --network to the following network configuration
 - a. First two to: alpine-net
 - b. 3rd one to the **default bridge**
 - c. 4th one to **alpine-net & to the bridge network (trickey...)**

Tip: network attach...

4. Inspect Network bridge and user defined network

User Define Bridge network

LAB:

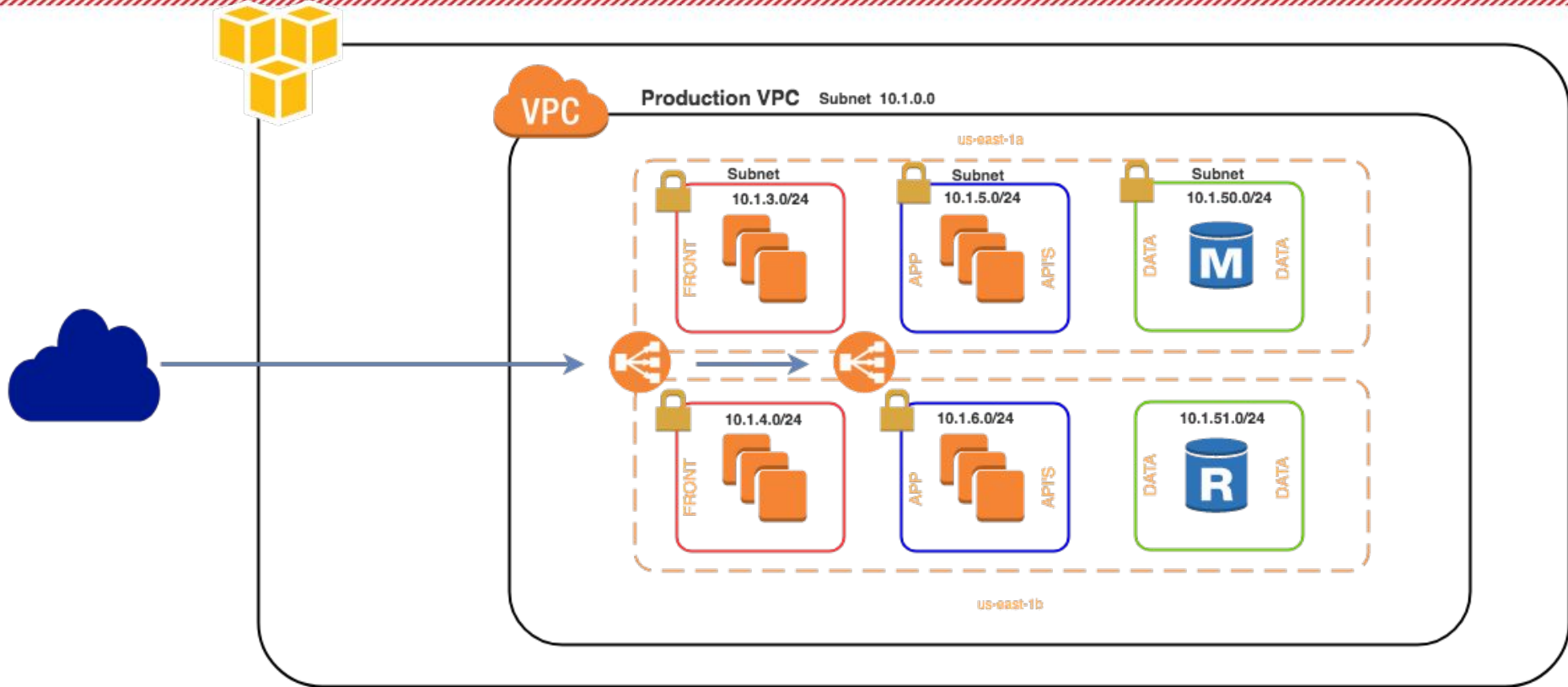
5. Connect to alpine1 and try pinging to alpine1,2,3,4 with IP and DNS - What happened ?
6. Connect to alpine4 and try pinging to alpine1,2,3,4 with IP and DNS - What happened ?
7. Why?
8. Stop all containers , Remove them and delete the user defined network you created



NETWORK BRIDGES

LAB: CONNECTING APP AND DB

Network



APP + DB Network layer and Source code hook

LAB:

1. Create two network bridges
 - a. db_layer
 - b. App_layer
2. Copy spring-music.jar from `/seminars/docker/artifacts` to a new folder of your choose.
3. Run MYSQL Container as followed and with the following: (1 line)

```
docker run --rm -itd --name db_mysql -e MYSQL_ALLOW_EMPTY_PASSWORD=yes -e  
MYSQL_DATABASE=music wangxian/alpine-mysql
```

And add the following:

4. Mount: `"$(pwd)"/:/app`
5. Networks: `db_mysql`
6. Expose port: `3306:3306`
7. Name: `db_mysql`

9. Inspect db_mysql

- Verify MYSQL is working by connecting to the container and running mysql
- Verify a new local folder on your host called mysql created
- Inspect network and container that it indeed connected to db_layer bridge

10. Create new java web application container that will run your local spring-music jar

- Image: `frolvlad/alpine-oraclejdk8:slim`
- CMD: `java -jar -Dspring.profiles.active=mysql /source/spring-music.jar`
- Mount: `"$(pwd)":/source`
- Networks: `app_mysql & db_mysql`
- Expose port: `8080:8080`
- Name: `web_app`

11. Validate application is working with the DB container in db_layer

- a. Connect to db_mysql and run:
 - i. “Mysql” and select database “music” and view table “albums”
- b. Browse <http://localhost:80> and change a value and than review table “albums” again
 - i. issues ?
 1. Check logs using docker logs [container]
 2. Make sure web_app is connected to both db_layer and app_layer
 3. Inspect network and container

12. Once everything is working -

- a. stop containers & delete bridges.



Docker Advanced

Dockerfile - Custom images

Module Agenda

“Docker can build images automatically by reading the instructions from a Dockerfile. A Dockerfile is a text document that contains all the commands a user could call on the command line to assemble an image. Using docker build users can create an automated build that executes several command-line instructions in succession “

Getting started with Dockerfile

Creating a Custom Dockerfile

Building a Custom image

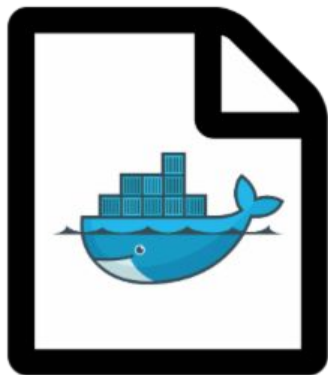
Publishing an Image to Docker
Hub

What will we do in this module?

Get our source code into a **custom built image (vs pre-built images)** to share with others

What is a dockerfile and how it create an Image

Developers use .java or pom file to describe / develop - we use Dockerfile



Dockerfile

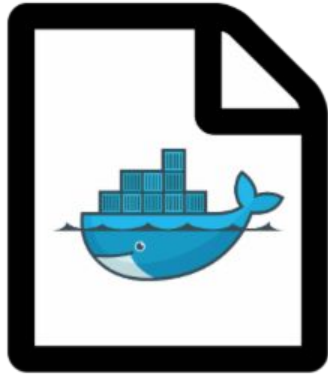


pom.xml



Hello.java

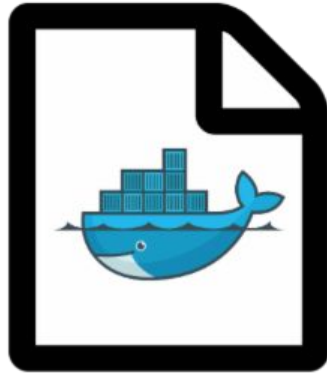
Dockerfile is a ... FILE with instructions and descriptions of an image



Dockerfile

BUILD	Boot	Run
FROM	WORKDIR	CMD
MAINTAINER	USER	ENV
COPY		EXPOSE
ADD		VOLUME
RUN		ENTRYPOINT
ONBUILD		
.dockerignore		

Dockerfile flow



dockerfile



Build - done by Docker
daemon and not CLI



image

Dockerfile flow Overview

Text file with instructions

Build process sends entire context
to daemon recursively

```
$ docker build -t [repository:tag] .  
Sending build context to Docker daemon 6.51 MB  
...
```

Docker IMAGE created

Warning: Do not use your root directory, /, as the PATH as it causes the build to transfer the entire contents of your hard drive to the Docker daemon.



Dockerfile - Example

```
ARG VERSION=latest
# Java8 Alpine Release
FROM frolvlad/alpine-oraclejdk8:slim
ARG VERSION
# RUN will execute shell commands
RUN echo $VERSION > image_version
# Label Use Labels for descriptions and view it with docker inspect
LABEL multi.label1="value1" \
    description="Bug fix x.0 for client y"
# configure WorkDir inside the container
WORKDIR /app
# Mount HOST Folder
VOLUME ["/spring-boot-rest-example/dockerfile/artifact/"]
# Copy Spring Boot File to target
COPY spring-boot-rest-example-0.4.0.war /app/spring-boot-rest-example-0.4.0.war
#Expose Ports - ONLY EXPOSED - IT'S NOT Mapped. -p will be needed on run
EXPOSE 8091
EXPOSE 8090
#The HEALTHCHECK instruction tells Docker how to test a container to check that it is still working
HEALTHCHECK --interval=5m --timeout=3s \ CMD curl -f http://localhost/ || exit 1
# The main purpose of a CMD is to provide defaults for an executing container
CMD java -jar -Dspring.profiles.active=test /app/spring-boot-rest-example-0.4.0.war
```


CMD VS ENTRYPOINT?

```
FROM alpine:latest  
CMD ping localhost
```

```
docker build -t playground:latest .  
....  
docker run -ti playground:latest  
PING localhost (127.0.0.1): 56 data bytes  
64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.051 ms  
64 bytes from 127.0.0.1: seq=1 ttl=64 time=0.080 ms  
# in CMD - Override IS ALLOWED  
#### docker run -ti playground:latest [command]  
docker run -ti playground:latest hostname  
93d4a120e1ff
```

CREATING A CUSTOM BOOTSPRING DOCKERFILE

MAKE SURE YOU CLONED

<https://github.com/yanivomc/seminars.git>

1. Make a new folder in your project directory called `jb_dockerfile`
 - a. Copy `spring-music.jar` from `/seminars/docker/artifacts` to a new folder `jb_dockerfile/artifacts`
 - b. Create an empty `dockerfile`
2. SPEC
 - a. From: `frolvlad/alpine-oraclejdk8:slim`
 - b. Workdir `/data`
 - c. Copy: artifact to `/app`
 - d. Expose: `8080:8080`
 - e. CMD: `java -jar -Dspring.profiles.active /app/spring-music.jar`
3. Build & Run image

Building the dockerfile in CLI

```
# Build dockerfile
# docker build -t [repo/imagename:tag] [dockerfile location]

# Run image created above
docker run -p [port_source:port_targe] --rm -ti --name [container name] [image]:tag
```

Browse

<http://localhost:8080/>

DOCKER HUB

Push our docker image to docker hub

1. Create new Repo in docker hub
2. Register your newly created repo and login to it in CLI
“docker login”
3. Push your created image to your repo
“docker push repo/image:tag”



Maven style

Building and Pushing Docker image to automate build process

MAVEN & DOCKER

Using [Spotify Maven Plugin](#), Build , Deploy and Push Docker Image post build becomes extremely easy

Once configured we can run: mvn clean package docker:build

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.spotify.it</groupId>
  <artifactId>boot</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <description>The Dockerfile is built, and later put into a repository</description>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.dockerArtifactId>jb-springboot-example</project.dockerArtifactId>
  </properties>

  <build>
```

```
<build>
  <resources>
    <resource>
      <!-- <directory>src/main/resources</directory> -->
      <directory>artifact</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
  <!-- Docker Build -->
<plugins>
  <plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.0.0</version>
  </plugin>
  <plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.10</version>
    <configuration>
      <imageName>yanivomc/${project.dockerArtifactId}</imageName>
      <dockerDirectory>dockerfile</dockerDirectory>
      <resources>
        <resource>
          <targetPath>.</targetPath>
          <directory>${project.build.directory}</directory>
        </resource>
      </resources>
    </configuration>
  </plugin>
</plugins>
</build>
<include>${project.artifactId}-${project.version}.${project.packaging}</include>
```



Docker Advanced

Docker Compose

Intro

“ Compose is a tool for defining and running multi-container Docker applications...

.... With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration. “

MODULE AGENDA

Getting started

Docker-compose.yml

Docker compose commands

Docker compose in action

Setting up dev env services

Creating custom docker-compose
to manage our services

When should we use it?

Development environment

When you're developing software, the ability to run a full fledged application (role and all of its dependencies) in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

Using the Compose file - we can document and configure all of the application (role) dependencies (DB, Queues, Caches, web services API and many other components) in one of multiple containers per component in a **single command** (docker-compose up)

Compose can provide a convenient way for developers to focus on developing and not on requesting servers or waiting for IT to provide VM's , EC2's or physical servers to develop on top.

When should we use it?

Automated Tests environment (as part of a ci/cd or standalone)

With compose we can run end-to-end testing that requires a full environment for it to run.

Compose provides a convenient way to create , destroy an isolated testing environments for our test suite.

Vision this:

```
|$ docker-compose up -d  
$ ./run_ui_test  
$ docker-compose down
```


Docker compose

When should we use it?

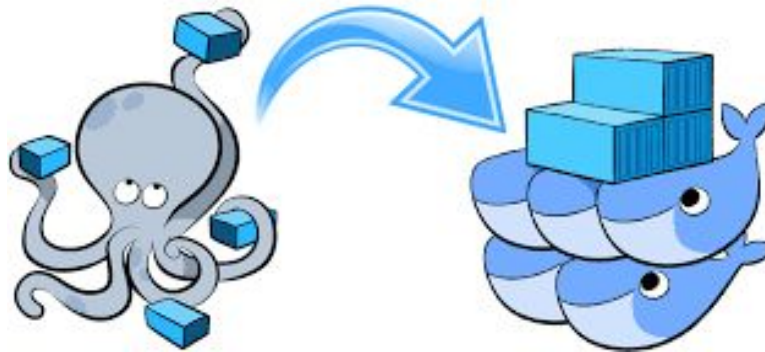
Production use....

Possibly but we got Kubernetes for that purpose (and no... we are not learning about K8S today...)

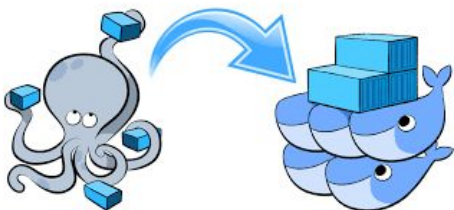
Docker Compose

Layout

Docker compose manages our application lifecycle



Docker compose manages our application lifecycle

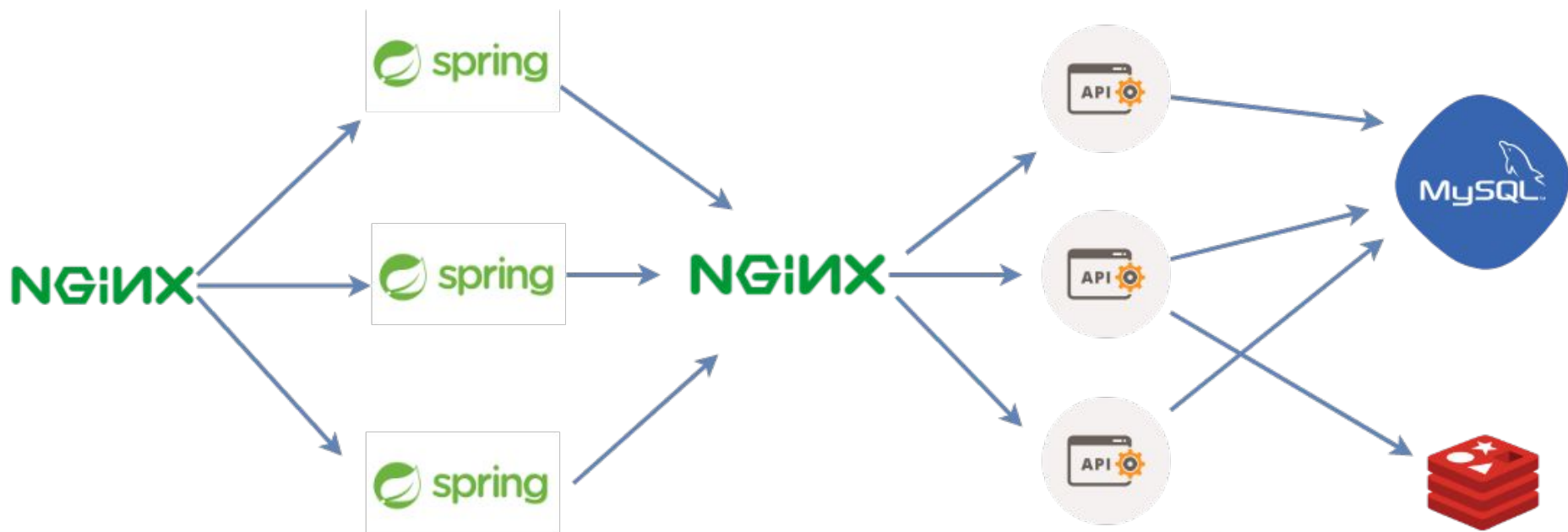


- Start, stop , rebuild of our services
- View status of our running services
- Stream the log output of running services
- Run a one-off command on a service

Docker compose

Why do we need docker compose?

Imagine managing this manually



Why do we need docker compose?

Using `docker-compose.yml` we can define

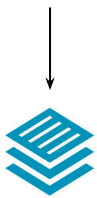
- Networking
- Dependencies between services
- Environments
- What makes up a role and its components
- Manage each application services we got

Docker compose

Docker compose flow



Build Services
(dockerfiles)



Generate images



Start our services



Take them down

Layout

Docker compose is basically 3 steps process

1. First we define our app/role environment with a dockerfile as we did earlier
2. Define the services/components that makes our app/role a whole in our compose file
“docker-compose.yml”
3. Run “docker-compose up”
4. Multiple images are up and running



Docker Compose

.yml example



Compose file version 3 reference

<https://docs.docker.com/compose/compose-file/>

Common cli for docker compose lifecycle

```
build    — Build or rebuild services
bundle   — Generate a Docker bundle from the Compose file
config   — Validate and view the Compose file
create   — Create services
down     — Stop and remove containers, networks, images, and volumes
events   — Receive real time events from containers
exec     — Execute a command in a running container
help     — Get help on a command
images   — List images
kill     — Kill containers
logs     — View output from containers
pause    — Pause services
port     — Print the public port for a port binding
ps       — List containers
pull     — Pull service images
push     — Push service images
restart  — Restart services
rm       — Remove stopped containers
run      — Run a one-off command
scale    — Set number of containers for a service
start    — Start services
stop     — Stop services
top      — Display the running processes
unpause  — Unpause services
up       — Create and start containers
version  — Show the Docker-Compose version information
```



Docker Compose

Follow Through

Simple Project with docker compose

Project Description:

We will build and run an application with two roles.

Front: Web - our spring-music spring app using the dockerfile we created

Backend: db_sql - mysql db using the dockerfile we created

Step 1 - prep

- Create new folder name: `jb_dockercompose`
- Copy your mymusic-spring dockerfile we created earlier
 - or from `seminars/docker/playground/labs/labdockercompose/roles/web/dockerfile`
 - Into: `jb_dockercompose/roles/web/dockerfile`
- Update the location of the artifact `spring-music.jar` in the dockerfile
 - You can use the one in: `seminars/docker/artifacts/spring-music.jar`

Step 2 - WEB Roles - Dockerfile

```
FROM frovlad/alpine-oraclejdk8:slim
WORKDIR /code
EXPOSE 8080:8080
CMD java -jar -Dspring.profiles.active /code/spring-music.jar
```


Step 4 - Build and run the app with compose

Run `docker-compose build` and then `docker-compose up -d`

Try:

- `docker-compose logs -f`
- `docker-compose ps`

```
yanivos@yanivos > ~/work/repos/seminars/docker/playground/labs/labdockercompose > master > docker-compose up
Starting labdockercompose_db_mysql_1 ...
Starting labdockercompose_db_mysql_1 ... done
Recreating labdockercompose_web_1 ...
Recreating labdockercompose_web_1 ... done
Attaching to labdockercompose_db_mysql_1, labdockercompose_web_1
db_mysql_1 | [i] MySQL data directory not found, creating initial DBs
db_mysql_1 | 2018-03-03 23:16:32 140561907043112 [Note] /usr/bin/mysqld (mysqld 10.1.19-MariaDB) starting as process 35 ...
db_mysql_1 | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Using mutexes to ref count buffer pool pages
db_mysql_1 | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: The InnoDB memory heap is disabled
db_mysql_1 | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Mutexes and rw_locks use GCC atomic builtins
db_mysql_1 | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: GCC builtin __atomic_thread_fence() is used for memory barrier
db_mysql_1 | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Compressed tables use zlib 1.2.8
db_mysql_1 | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Using Linux native AIO
db_mysql_1 | 2018-03-03 23:16:32 140561907043112 [Note] InnoDB: Using SSE crc32 instructions
```

Docker compose

Browse <http://localhost:8080>

localhost:8080

marks ☒ חשבונות ירוקה | Online learning | Team A | FaceBook | News | Training links | Resources | Website | Seminars links | Clients | Apps | 58f881147dcd0-gdp... | Other Bo

Spring Music 🎵

Albums

[view as: | sort by: title artist year genre ^ | +add an album]

London Calling
The Clash

Don't Look
Back

Rock With Me
The Fabulous

Achtung Baby
U2



CLEAN UP

`docker-compose rm -f -v -s`



Final Lab

Maven and docker compose

Project description

LAB 1.0

Roles:

Front: Web - with Spring Boot jar demo

Backend: redis - Redis DB

Description:

1. Create new Project folder
2. Copy **/seminars/docker/dockercompose/artifacts/boot-0.0.1-SNAPSHOT.jar** to your project folder
3. Create new docker file with img : **frolvlad/alpine-oraclejdk8:slim & CMD**
“java -jar -Dspring.profiles.active=test /code/boot-0.0.1-SNAPSHOT.jar”
4. Create a docker-compose file that build the web and db + exposing port 8080 and map local “.” to /code inside the web container

Project description

LAB 2.0

Roles:

Front: Web - Create your own spring web project (simple) that connect to Redis and use maven to build and create and artifact

Backend: redis - Redis DB

Description:

1. Create new Project folder
2. Make sure maven build and deploy the artifact to deploy into the project folder
3. Create new docker file with img : **frolvlad/alpine-oraclejdk8:slim & CMD**
"java -jar -Dspring.profiles.active=test /code/{YOUR-ARTIFACT.jar}"
4. Create a docker-compose file that build the web and db + exposing your application port and map local "." folder to /code inside the web container for CD.
5. Change your code, rebuild it and rebuild your docker-compose to see your changes in live.



Where do we go next?

Where to go next ?

Type	Software
Clustering/orchestration	Swarm, Kubernetes, Marathon, MaestroNG, decking, shipyard
Docker registries	Portus, Docker Distribution, hub.docker.com, quay.io, Google container registry, Artifactory, projectatomic.io
PaaS with Docker	Rancher, Tsuru, dokku, flynn, Octohost, DEIS
OS made of Containers	RancherOS



QUESTIONS ?

END

DevOps Course

JOHN BRYCE
Leading in IT Education
matrix company