

# 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/>



# 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...

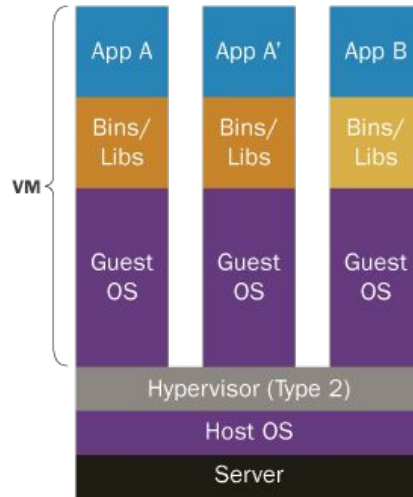
- What Do You Know About Docker?
- Who Used Docker For Development / QA / STG / PROD?
- Who Tried & Failed Implementing Docker

# What is Docker

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

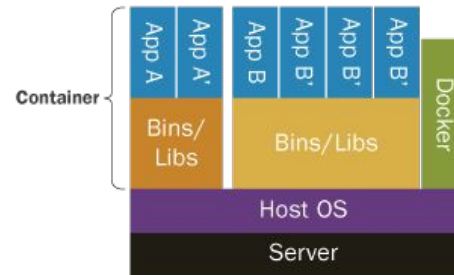
Docker allows you to package an application with all of its dependencies into a standardized unit for software development.

# 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

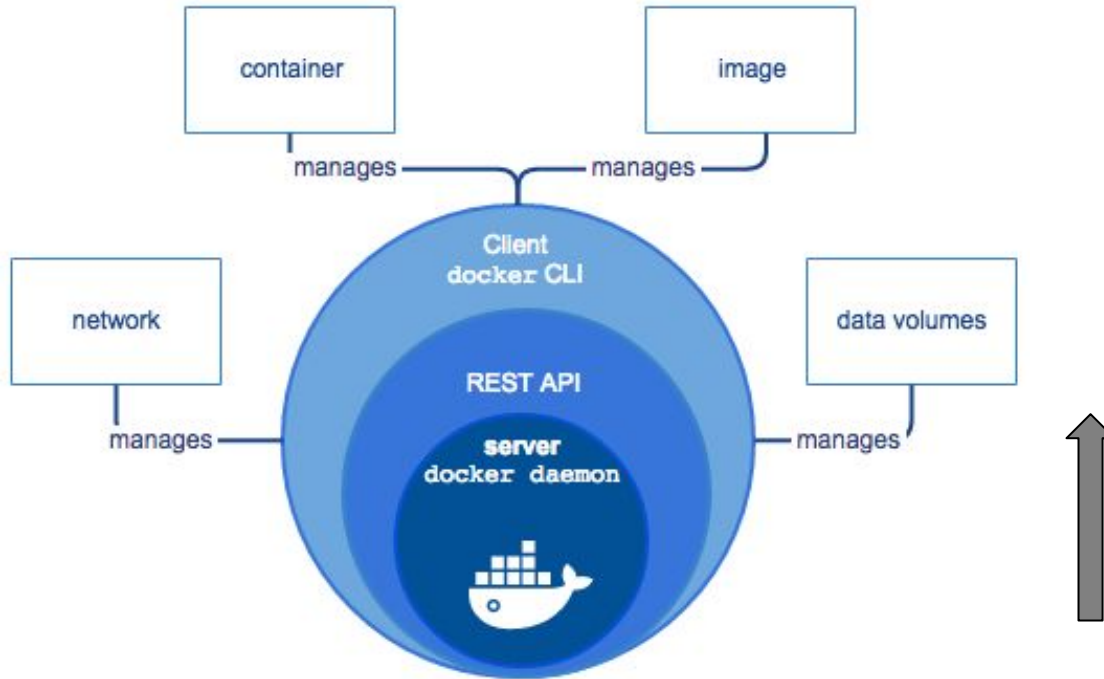
# Common Use Cases for Docker

- 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



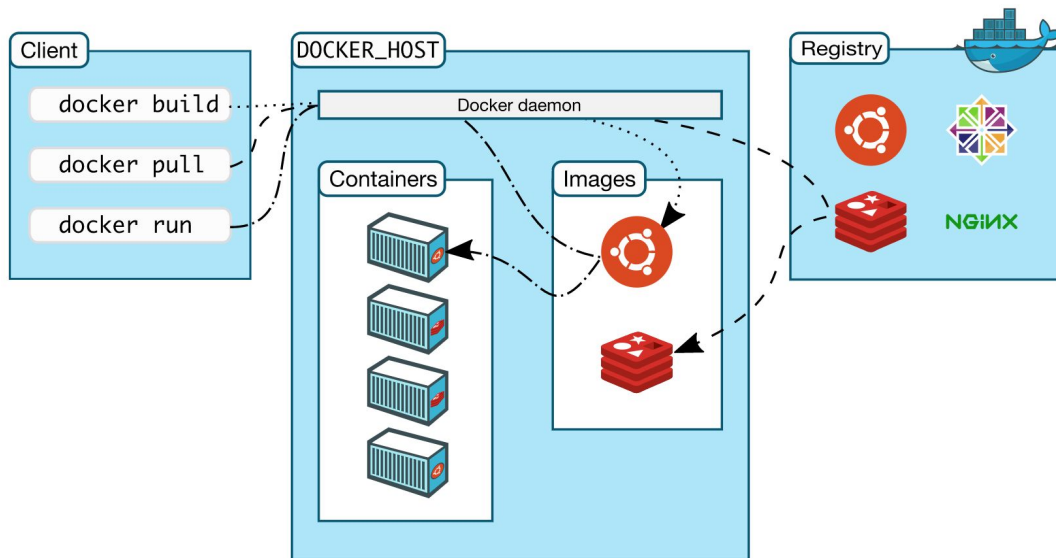
# What is docker - Technical Aspect

## 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

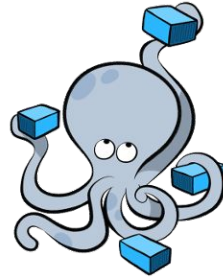
## Docker Objects

- 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

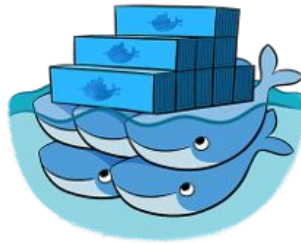
- Services
  - a. Allow you to scale containers across multiple Docker daemons, which all work together as a swarm with multiple managers and workers. Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API. A service allows you to define the desired state, such 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
98debfd4458	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

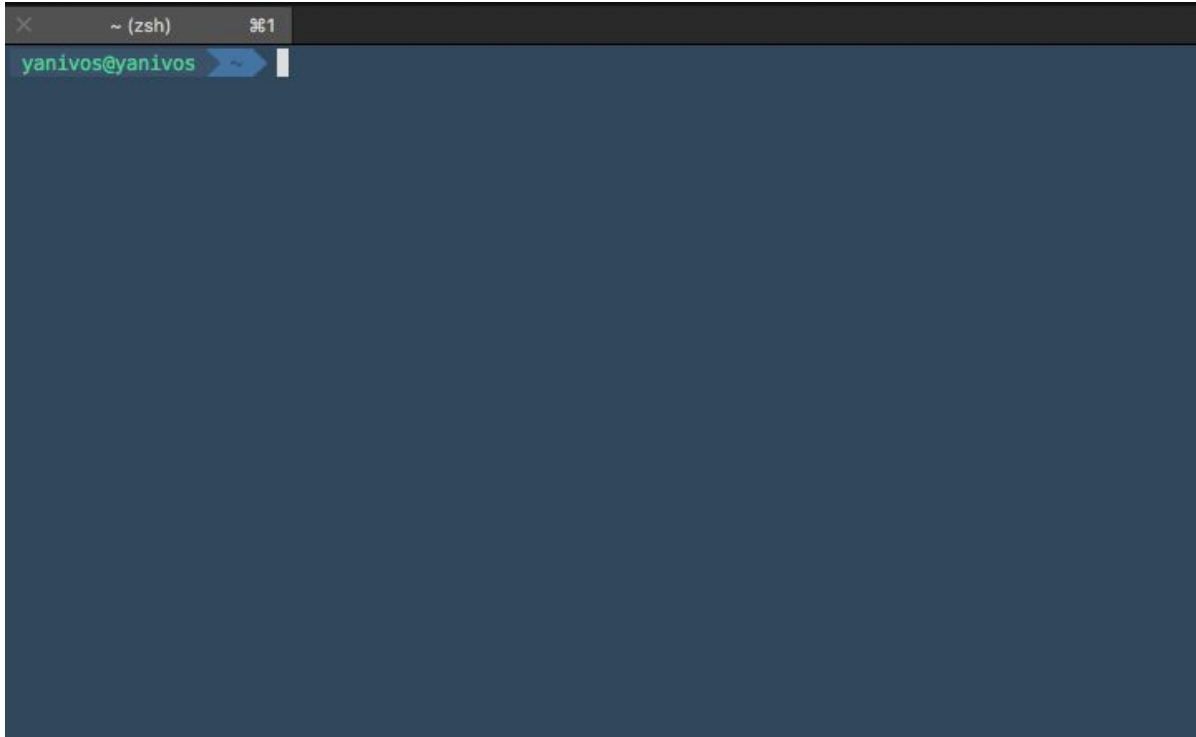
# Docker Examples

- 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

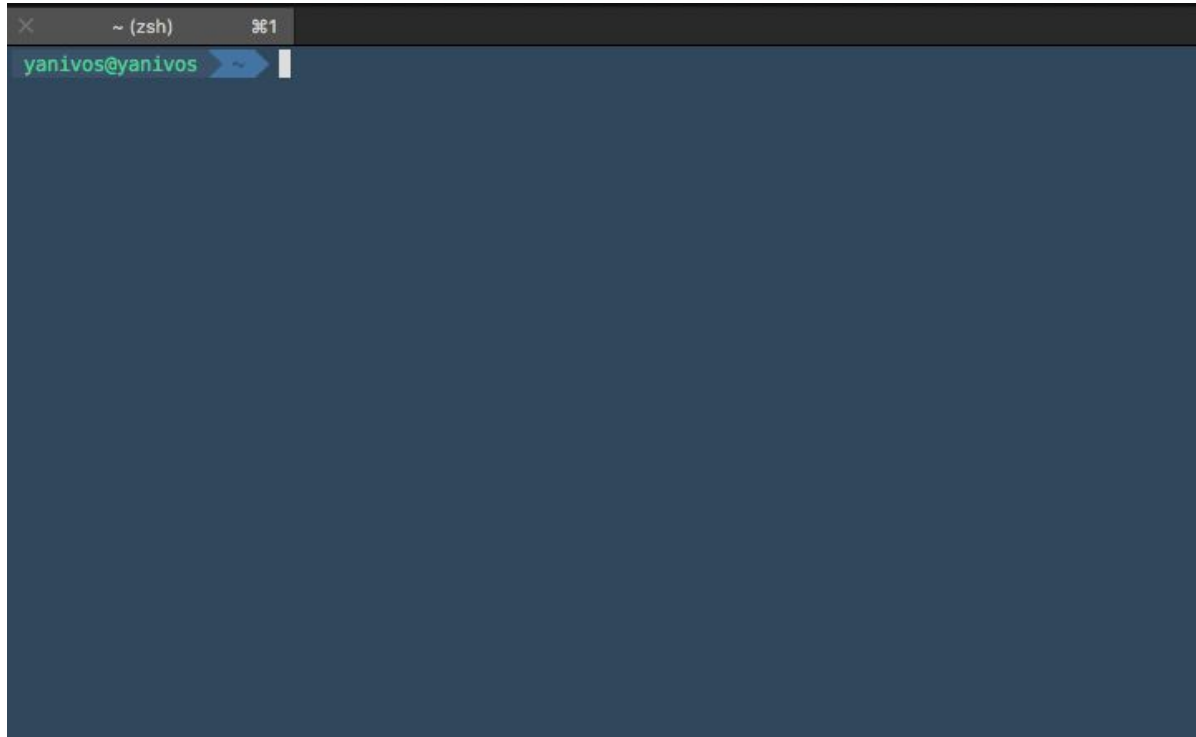
```
// General info
man docker // man docker-run
docker help // docker help run
docker info
docker version
docker network ls
// Images
docker images // docker [IMAGE_NAME]
docker pull [IMAGE] // docker push [IMAGE]
// Containers
docker run
docker ps // docker ps -a, docker ps -l
docker stop/start/restart [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.



# Building Our Own Jar Application

## In a Docker Container

# Building Our Own Jar Application

## In a Docker Container

For this example we will build a simple SpringBoot-rest application downloaded from [here](#)

Later in your LAB you will build your own SpringBoot artifact base on what you learned.



# Building Our Own Jar Application

## Creating sample Jar

```
## Java Class
public class HelloWorld {
    public static void main(String[] args){
        System.out.println("Hello World :) ");
    }
}
```

```
# Compile
javac HelloWorld.java
```

```
# Create manifest.txt
```

```
Manifest-Version: 1.0
Created-By: Me
Main-Class: HelloWorld
```

```
# Create Jar file
```

```
jar cfm HelloWorld.jar manifest.txt HelloWorld.class
```

```
# Check if working
```

```
java -jar HelloWorld.jar
```

# Spring Boot Application On Docker

## Creating dockerfile with a demo war / jar

```
# Java8 Alpine Release
FROM frovlad/alpine-oraclejdk8:slim

# configure WorkDir
WORKDIR /app

# Mount HOST Folder
VOLUME /app

# Copy Spring Boot File to target
COPY jar/spring-boot-rest-example/target/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

# Command to run (Entry Point)
CMD java -jar -Dspring.profiles.active=test /app/spring-boot-rest-example-0.4.0.war
```

# Spring Boot Application On Docker

## Building the dockerfile in CLI

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

# Run image created above
docker run -p 8091:8091 -p 8090:8090 --rm -t -i --name springboot
yanivomc/dockerspringboot:latest
```

# Spring Boot Application On Docker

## Running the Image

```
# Command to run (Entry Point)
docker run -p 8091:8091 -p 8090:8090 --rm -t -i --name springboot yanivomc/dockerspringboot:latest
```

```
framework.boot.actuate.endpoint.mvc.EndpointMvcAdapter.invoke()
2018-01-13 22:52:12.211 INFO 7 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "[{/error}]" onto public java.util.Map<
2018-01-13 22:52:12.217 INFO 7 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of t
2018-01-13 22:52:12.217 INFO 7 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [cla
2018-01-13 22:52:12.231 INFO 7 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework
arent: org.springframework.boot.context.embedded.AnnotationConfigEmbeddedWebApplicationContext@4aa8f0b4
2018-01-13 22:52:12.318 INFO 7 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8091 (http)
2018-01-13 22:52:12.323 INFO 7 --- [           main] o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 2147483647
2018-01-13 22:52:12.324 INFO 7 --- [           main] d.s.w.p.DocumentationPluginsBootstrapper : Context refreshed
2018-01-13 22:52:12.341 INFO 7 --- [           main] d.s.w.p.DocumentationPluginsBootstrapper : Found 1 custom documentation plugin(s)
2018-01-13 22:52:12.351 INFO 7 --- [           main] s.d.s.w.s.ApiListingReferenceScanner : Scanning for api listing references
2018-01-13 22:52:12.517 INFO 7 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8090 (http)
```

# 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” >> “docker push yanivomc/learningdocker:latest”



# 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

## Network & Docker Compose

## 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

## 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 -dit --name alpine1 alpine ash  
docker run -dit --name alpine2 alpine ash
```

As you recall: The `-dit` 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 ?

# Network - Basics

## Inspect example

```
yanivos@ip-10-0-0-25 ➤ docker network inspect bridge
[
  {
    "Name": "bridge",
    "Id": "544e9ba6b7dd00829afab0c8599ca78f5dcfa07db93893d730185b2d9ccd9ca4",
    "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": "7809b16709d0ae7a752b5d3e50272d1358347ad51b4ada1d0c49b7bfbeb1da4e",
        "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.

## User Define Bridge network

### LAB:

1. Delete the previous containers (stop and then remove)
2. Create a newly created 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

# Where to go next Network?

Want to Learn more about docker network?

follow online tutorials for:

- **Host networking** driver (only works on linux stations)
- Overlay networking with swarm (clustering)



# 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. “

## 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....

We basically can but we got Kubernetes for that purpose



# Docker Compose

## Layout

## 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” and all done.

# Docker compose

## Ex. of docker-compose file

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - ./code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

## 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

## Getting things done with docker compose

### 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” and all done.

## Simple Project with docker compose

### Project Description:

We will build and run an application with two roles.

**Front:** Web - Simple flask Python application

**Backend:** redis - Redis DB

## Step 1 - Application Code

- Simple python code named app.py under **../seminars/docker/dockercompose**
- Python Requirements file named “requirements.txt”

## Step 2 - Docker File

```
# Build an image starting with the Python 3.4
FROM python:3.4-alpine
#Add the current directory . into the path /code in the image
ADD . /code
#Set the working directory to /code.
WORKDIR /code
#Install the Python dependencies
RUN pip install -r requirements.txt
#Set the default command for the container to python app.py.
CMD ["python", "app.py"]
```

## Step 3 - Define services in compose file

```
version: '3'
services:
  web:
    build: .
    ports:
      - "5000:5000"
  redis:
    image: "redis:alpine"
```

- This Compose file defines two services, web and redis.
- The web service: Uses an image that's built from the Dockerfile in the current directory. Forwards the exposed port 5000 on the container to port 5000 on the host machine. We use the default port for the Flask web server, 5000.

## Step 4 - Build and run the app with compose

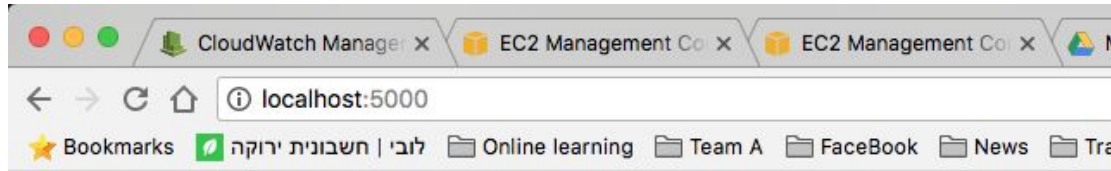
Run docker-compose up

```
yanivos@ip-10-0-0-25 ~/work/repos/learningDocker/repo/seminars/docker/dockercompose } master docker-compose up
Recreating dockercompose_web_1 ...
Starting dockercompose_redis_1 ...
Recreating dockercompose_web_1
Recreating dockercompose_web_1 ... done
Attaching to dockercompose_redis_1, dockercompose_web_1
redis_1 | 1:C 20 Feb 23:11:27.622 # 000000000000 Redis is starting 000000000000
redis_1 | 1:C 20 Feb 23:11:27.622 # Redis version=4.0.8, bits=64, commit=00000000, modified=0, pid=1, just started
redis_1 | 1:C 20 Feb 23:11:27.622 # Warning: no config file specified, using the default config. In order to specify a config file use redis-server /path/to/redis.conf
redis_1 | 1:M 20 Feb 23:11:27.623 * Running mode=standalone, port=6379.
redis_1 | 1:M 20 Feb 23:11:27.623 # WARNING: The TCP backlog setting of 511 cannot be enforced because /proc/sys/net/core/somaxconn is set to the lower value of 128.
redis_1 | 1:M 20 Feb 23:11:27.623 # Server initialized
redis_1 | 1:M 20 Feb 23:11:27.623 # WARNING you have Transparent Huge Pages (THP) support enabled in your kernel. This will create latency and memory usage issues with Redis.
root, and add it to your /etc/rc.local in order to retain the setting after a reboot. Redis must be restarted after THP is disabled.
redis_1 | 1:M 20 Feb 23:11:27.624 * DB loaded from disk: 0.000 seconds
redis_1 | 1:M 20 Feb 23:11:27.624 * Ready to accept connections
web_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
web_1 | * Restarting with stat
web_1 | * Debugger is active!
web_1 | * Debugger PIN: 299-379-742
```



# Docker compose

Browse <http://localhost:5000>



Hello World! I have been seen 25 times.





# 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*