An overview of some container-based technologies

Containerization workshop

Infrastructures for Cloud Computing and Big Data M

Dmitrij David Padalino Montenero

Academic Year 2018/2019

Topics

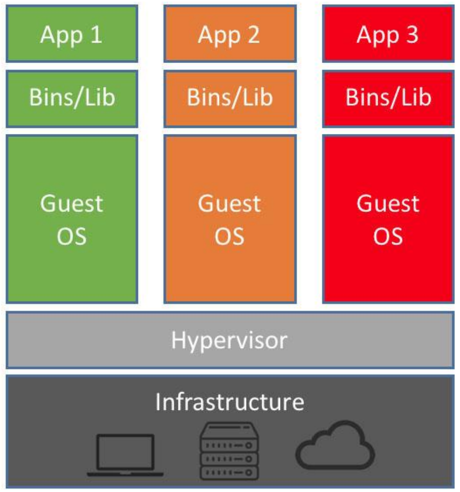
- Containerization and Orchestration
- Docker
- Kubernetes
- OpenShift
- IBM Cloud Kubernetes-as-a-Service

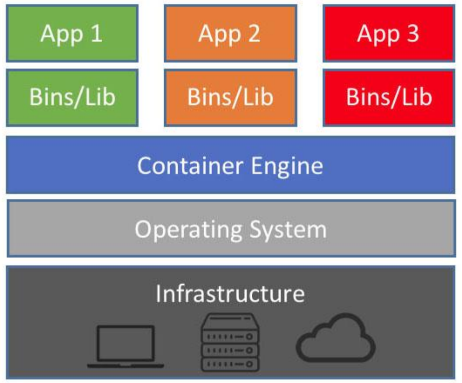
Containerization

Containerization refers to an operating system feature in which the kernel allows the existence of multiple isolated user-space instances.

Such instances, called **containers**, partitions or virtual environments may look like real computers from the point of view of programs running in them.

Virtual Machines vs Containers - 1

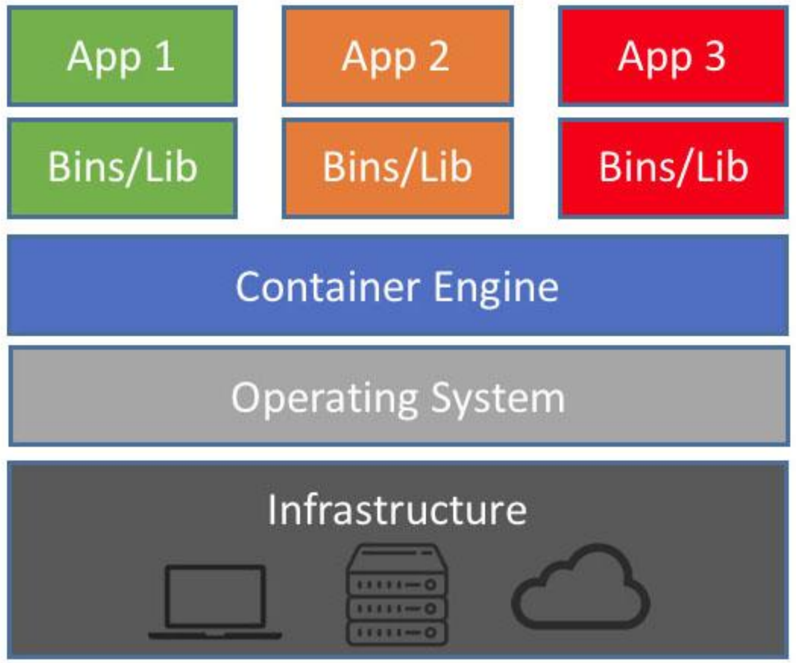




- Hypervisor
- Many guest OSs in the same physical server
- Visibility of all hardware resources
- Well separated run-time environments
- Images are Gigabyte in size
- Hardware resources optimization
- Cost savings

- Container engine
- Host OS kernel sharing
- Many containers in the same physical server
- Visibility only of the resources assigned to containers
- Relatively well separated run-time environments
- Images are Megabytes in size
- Further hardware resources optimization

Virtual Machines vs Containers - 2



Novel way to think about the architecture of a distributed application, so called **micro-services architecture**.

Each container is usually single-purpose, single-process, which aligns nicely with micro-services architectures.

The new approach is to package the different services that constitute an application into separate containers, and to deploy those containers across a cluster of physical or virtual machines.

Containers are

- Flexible
- Lightweight
- Interchangeable
- Portable
- Scalable

Orchestration

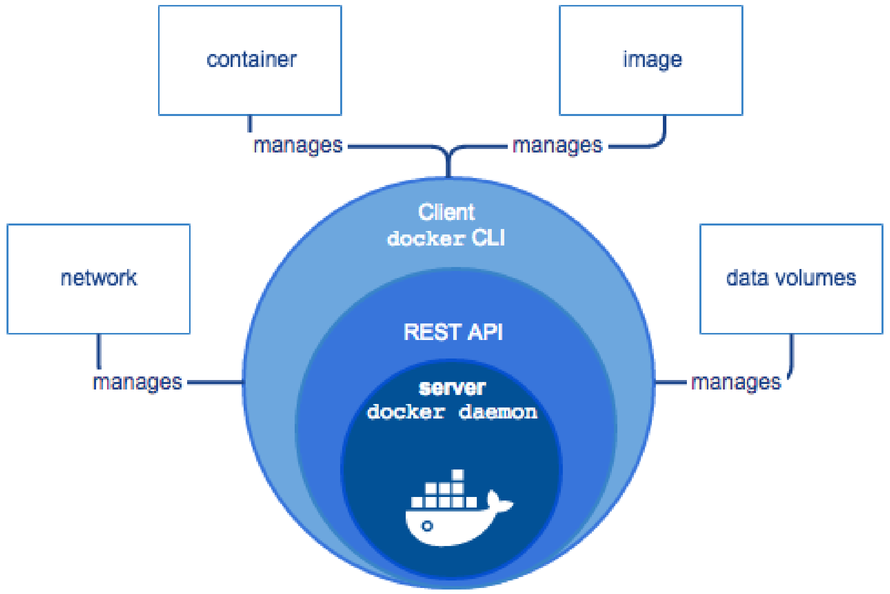
- Need to manage micro-services applications.

- **Container orchestration** allows users to control when containers start and stop, group them into clusters, and coordinate all of the processes that compose an application. Container orchestration tools allow users to guide container deployment and automate updates, health monitoring, and failover procedures.

Docker

Docker is a platform for developers and sysadmins to **develop**, **deploy**, and **run** applications with containers.

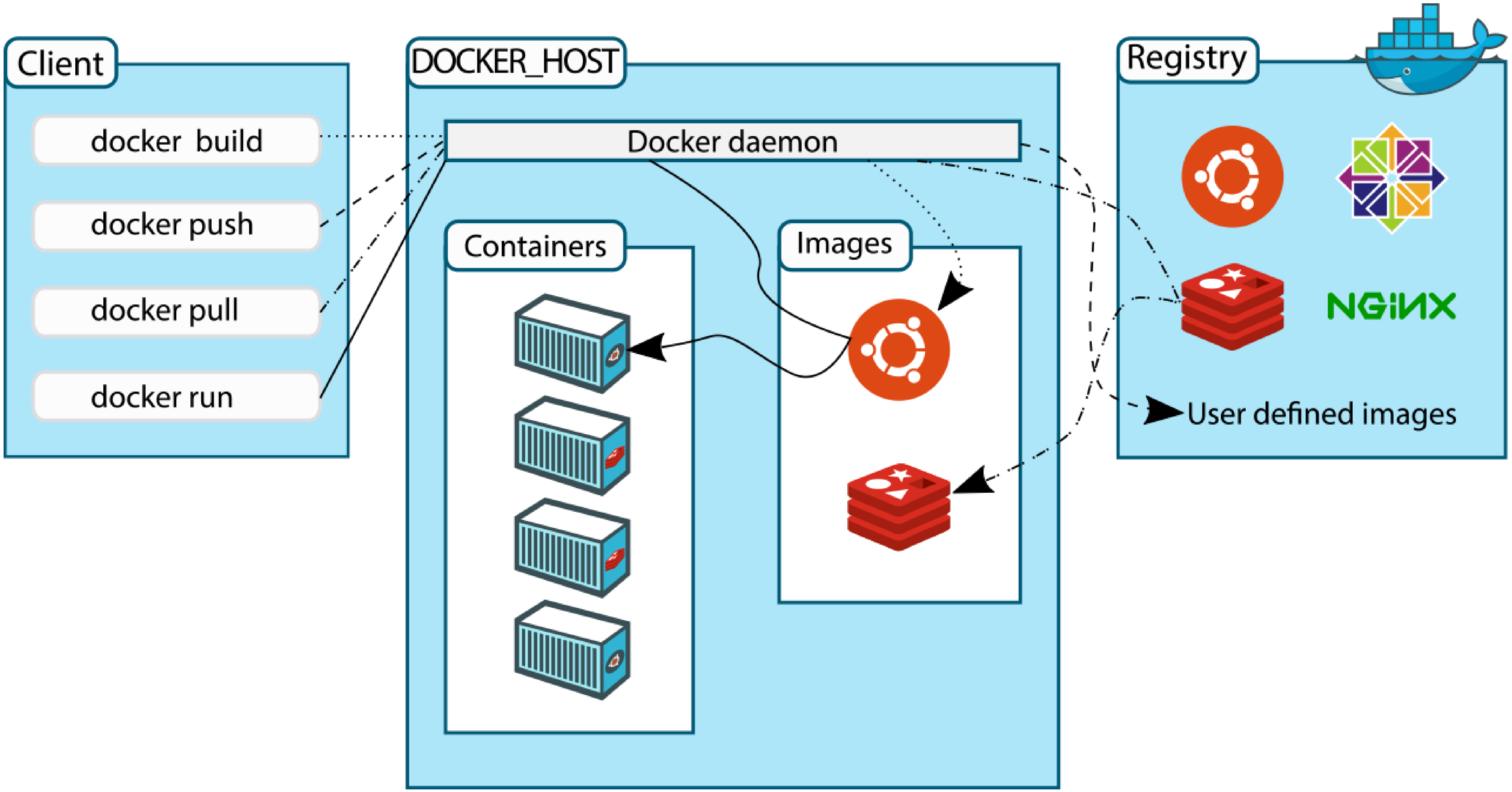
It enables you to separate your applications from your infrastructure so you can deliver software quickly.



Client-server architecture

- Server: docker daemon
- API REST
- Client: docker CLI

Architecture



Docker Lab

Starting from a 3-tier web application (Spring and MySQL)

- Application containerization
- Creation and configuration of a cluster (swarm) with Docker Machine.
- Deployment.
- Horizontal scaling.

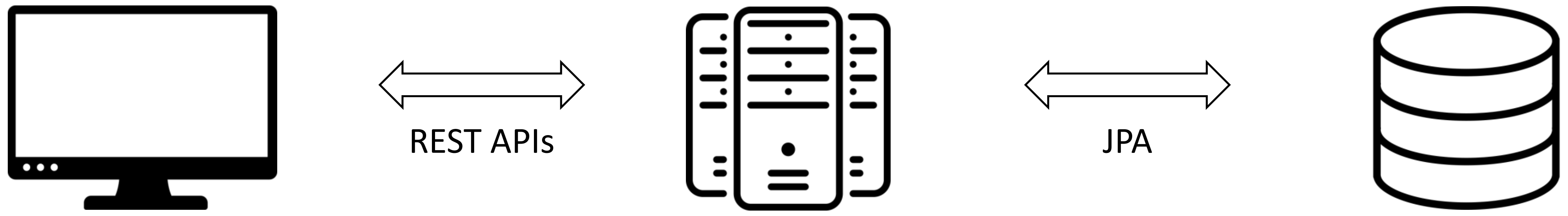
CPU	Intel(R) Core(TM) i7-3610QM CPU @ 2.30Ghz
RAM	8GB
SSD	Samsung 850 Evo 500GB
Interfaccia di rete wireless	Qualcomm Atheros AR5BWB22
Sistema operativo	Ubuntu 18.04.1 LTS Bionic Beaver
Versione Oracle VM VirtualBox	5.2.20r125813
Versione Docker	18.06.1-ce
Versione Docker Machine	0.14.0
Versione Google Chrome	69.0.3497.100 a 64 bit

The application

Source code link: https://github.com/davidMonnuar/projectActivity

3-tier web application for books collection management

Client Tier | Application Tier | Database Tier



- HTML5
- Javascript
- CSS

- Java
- Spring Boot
- Tomcat

- Rel. DB
- MySQL

Application Tier - 1

Spring Boot is an approach to develop spring application with minimal or zero configuration.

Tomcat used as an Embedded Server

Think about what you would need to be able to deploy your application (typically) on a virtual machine.

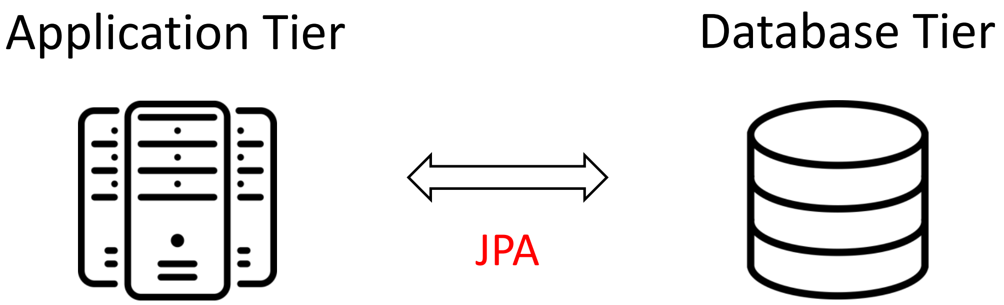
Step 1: Install Java
Step 2: Install the Web/Application Server (Tomcat/Websphere/Weblogic etc)
Step 3: Deploy the application war

What if we want to simplify this? This idea is the genesis for Embedded Servers.

For example, for a Spring Boot Application, you can generate an application jar which contains Embedded Tomcat. **You can run a web application as a normal Java application!**

Application Tier - 2

Thanks to **Spring deep integration with JPA** we don't need to spend too much time in configuring the persistence layer since most part of the setup is done automatically (i.e. db creation) by exploiting Java annotations.



The only thing we have to write down is a configuration file called **application.yml** in which we specify
-the URL to access the database
-username
-password
-plus other parameters

Entities
- Author.java
- Book.java

Repositories
- AuthorRepository.java
- BookRepository.java

Controller
- MainController.java

View
- CustomBookSerializer.java

Resources
- application.yml

Author.java - entities package

```java
package it.unibo.libraryDemo.entities;

import com.fasterxml.jackson.databind.annotation.JsonSerialize;
import it.unibo.libraryDemo.view.CustomBooksSerializer;

import javax.persistence.*;
import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;

@Entity
@Table(name="authors")
public class Author implements Serializable {
    @Id
    @Column(name="author_id")
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Integer id;

    private String name;

    @OneToMany(
            mappedBy = "author",
            cascade = CascadeType.ALL,
            orphanRemoval = true
    )
    @JsonSerialize(using = CustomBooksSerializer.class)
    private List<Book> books = new ArrayList<>();

    @PreRemove
    public void preRemove() {
        setBooks(null);
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public List<Book> getBooks() {
        return books;
    }

    public void setBooks(List<Book> books) {
        this.books = books;
    }

    public void addBook(Book book) {
        books.add(book);
        book.setAuthor(this);
    }

    public void removeBook(Book book) {
        books.remove(book);
        book.setAuthor(null);
    }
}
```

AuthorRepository.java - repositories package

```
package it.unibo.libraryDemo.repositories;

import it.unibo.libraryDemo.entities.Author;
import org.springframework.data.repository.CrudRepository;

public interface AuthorRepository extends CrudRepository<Author, Integer> {

}
```

CrudRepository is an interface and extends Spring data **Repository** interface.

CrudRepository provides generic CRUD operation on a repository for a specific type.

It has generic methods for CRUD operation.

To use CrudRepository we have to create our interface and extend CrudRepository. We need not to implement our interface, **its implementation will be created automatically at runtime**. Some methods we will use

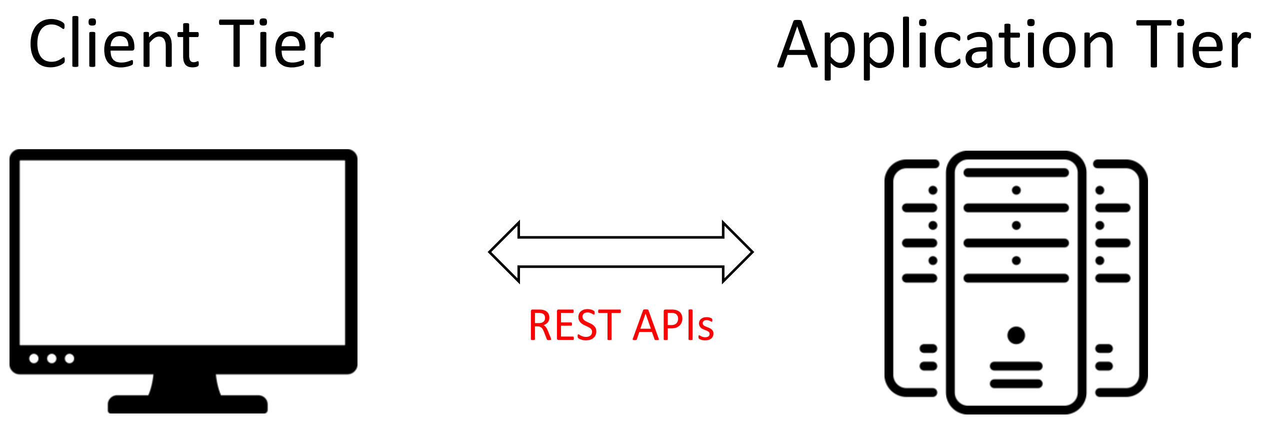
- findByID
- findAll
- deleteByID
- save

MainController.java - controller package - 1

In this class are defined and implementd the REST APIs that will be used by the HTML5 Client.

The APIs are

- library/getHostname
- library/resetApplication
- library/addBook
- library/getAllBooks
- library/deleteBook
- library/addAuthor
- library/getAllAuthors
- library/deleteAuthor



MainController.java - controller package - 2

```java
@RestController
@RequestMapping(path="/library")
public class MainController {
    @Autowired
    private AuthorRepository authorRepository;
    @Autowired
    private BookRepository bookRepository;

    @GetMapping(path="/getHostname")
    public String getHostname () {
        String hostname = "unknown";
        try {
            hostname = InetAddress.getLocalHost().getHostName();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
        return hostname;
    }

    @GetMapping(path="/resetApplication")
    public String resetApplication () {
        bookRepository.deleteAll();
        authorRepository.deleteAll();
        return "Done";
    }

    @PutMapping(path="/addBook")
    public String addNewBook (@RequestParam String title,
    @RequestParam String isbn, @RequestParam String author) {
        Book b = new Book();
        b.setTitle(title);
        b.setIsbn(isbn);
        Author a = authorRepository.
                   findById(Integer.parseInt(author)).orElse(null);
        if(a == null) {
            return "Not saved. The author is not in the database";
        }
        a.addBook(b);
        bookRepository.save(b);
        authorRepository.save(a);
        return "Book saved";
    }

    @GetMapping(path="/getAllBooks")
    public Iterable<Book> getAllBooks () {
        return bookRepository.findAll();
    }

    @DeleteMapping(path="/deleteBook")
    public String deleteBook (@RequestParam Integer id){
        bookRepository.deleteById(id);
        return "Book deleted";
    }

    @PutMapping(path="/addAuthor")
    public String addNewAuthor (@RequestParam String name) {
        Author a = new Author();
        a.setName(name);
        authorRepository.save(a);

        return "Author saved";
    }

    @GetMapping(path="/getAllAuthors")
    public Iterable<Author> getAllAuthors() {
        return authorRepository.findAll();
    }

    @DeleteMapping(path="/deleteAuthor")
    public String deleteAuthor (@RequestParam Integer id){
        authorRepository.deleteById(id);
        return "Author deleted";
    }
}
```

application.yml - resources package

We use global variables in order to avoid to hard-code the database information.

This let us to re-use this web app with any database we want.

```yaml
spring:
  profiles: container
  datasource:
    driverClassName: ${DATABASE_DRIVER}
    url: jdbc:mysql://${DATABASE_HOST}:${DATABASE_PORT}/${DATABASE_NAME}?useSSL=false&allowPublicKeyRetrieval=true
    username: ${DATABASE_USER}
    password: ${DATABASE_PASSWORD}
    tomcat:
      test-while-idle: true
      time-between-eviction-runs-millis: 60000
      validation-query: SELECT 1
  jpa:
    hibernate.ddl-auto: create-drop
    properties.hibernate.dialect: org.hibernate.dialect.MySQL5Dialect
```

Application containerization - 1

App organization

- One container for Spring
- One container for MySQL + Volume
- One overlay network for connecting the two containers

The first step is to define an image for each container by writing a text document called **Dockerfile**.

Thanks to a simple sintax a Dockerfile let us to define all the steps necessary to create an image and to execute it.

Application containerization - 2

Spring container Dockerfile

```
FROM openjdk:8-jre-alpine
MAINTAINER Dmitrij David Padalino Montenero

VOLUME /tmp
EXPOSE 8080

ARG JAR_FILE
ADD target/${JAR_FILE} app.jar
ADD wrapper.sh wrapper.sh

RUN apk add --update bash && rm -rf /var/cache/apk/*
RUN bash -c 'chmod +x /wrapper.sh'
RUN bash -c 'touch /app.jar'

ENTRYPOINT ["/bin/bash", "/wrapper.sh"]
```

Script entrypoint

```
#!/bin/bash
while ! exec 6<>/dev/tcp/${DATABASE_HOST}/${DATABASE_PORT}; do
echo "Trying to connect to MySQL at ${DATABASE_PORT}..."
sleep 10
done

java -Djava.security.egd=file:/dev/./urandom
-Dspring.profiles.active=container -jar /app.jar
```

Image creation and upload in the DockerHub

```
$ docker build -t librarydemo

$ docker login

$ docker tag librarydemo davidmonnuar/springbootlibrarydemo:1.0

$ docker push davidmonnuar/springbootlibrarydemo:1.0
```

Image size only 116 MB

MySQL container Dockerfile

No need to create a new Docker image since the official MySQL image available in the DockerHub is enough for our purposes.

Cluster creation and configuration

A **swarm** is a group of machines, physical or virtual, that are running Docker and joined into a cluster.

The cluster we will use is composed by two nodes, a master and one worker.

Virtual machines creation

```
$ docker-machine create –driver virtualbox myvm1

$ docker-machine create –driver virtualbox myvm2

$ docker-machine ls
NAME      ACTIVE    DRIVER      STATE    URL                          SWARM   DOCKER      ERRORS
myvm1     -         virtualbox  Running  tcp://192.168.99.100:2376            v18.06.1-ce
myvm2     -         virtualbox  Running  tcp://192.168.99.101:2376            v18.06.1-ce
```

Cluster initialization

```
$ docker-machine ssh myvm1 "docker swarm init --advertise-addr 192.168.99.100:2377
Swarm initialized: current node 0unmutpbeeeytxpquhiy1b6ok is now a manager.
To add a worker to this swarm, run the following command:
docker swarm join \
--token SWMTKN-1-4y8bpxyrbno89dapkkwyjdw3268qfzp128tpf5hecjdti5hb1k-00oatopl9dzw3jejbietxa9vp 192.168.99.100:2377

$ docker-machine ssh myvm2 "docker swarm join --token SWMTKN-1-4y8bpxyrbno89dapkkwyjdw3268qfzp128tpf5hecjdti5hb1k-
00oatopl9dzw3jejbietxa9vp 192.168.99.100:2377"

$ docker-machine ssh myvm1 "docker node ls"
ID                            HOSTNAME   STATUS   AVAILABILITY   MANAGER STATUS    ENGINE VERSION
0unmutpbeeeytxpquhiy1b6ok *   myvm1      Ready    Active         Leader            18.06.1-ce
ktvyi0nypz31l645iej8torkx     myvm2      Ready    Active                           18.06.1-ce
```

Deployment - 1

In a distributed application, different pieces of the app are called “services”.

Services in Docker are really just “containers in production.” A service only runs one image, but it codifies the way that image runs - what ports it should use, how many replicas of the container should run so the service has the capacity it needs, and so on. Scaling a service changes the number of container instances running that piece of software, assigning more computing resources to the service in the process.

In order to connect services Docker uses the concept of **Stack**. A stack is a group of interrelated services that share dependencies, and can be orchestrated and scaled together.

In the docker-compose.yml file are defined

- A service “ **mysqldb**”, port 3306
- A service “ **spring**” , port 80
- A service “ **visualizer**”, port 8080
- A volume “ **my-datavolume**” used by “mysqldb”
- A network “ **webnet**” with the default settings (which is a load-balanced overlay network)

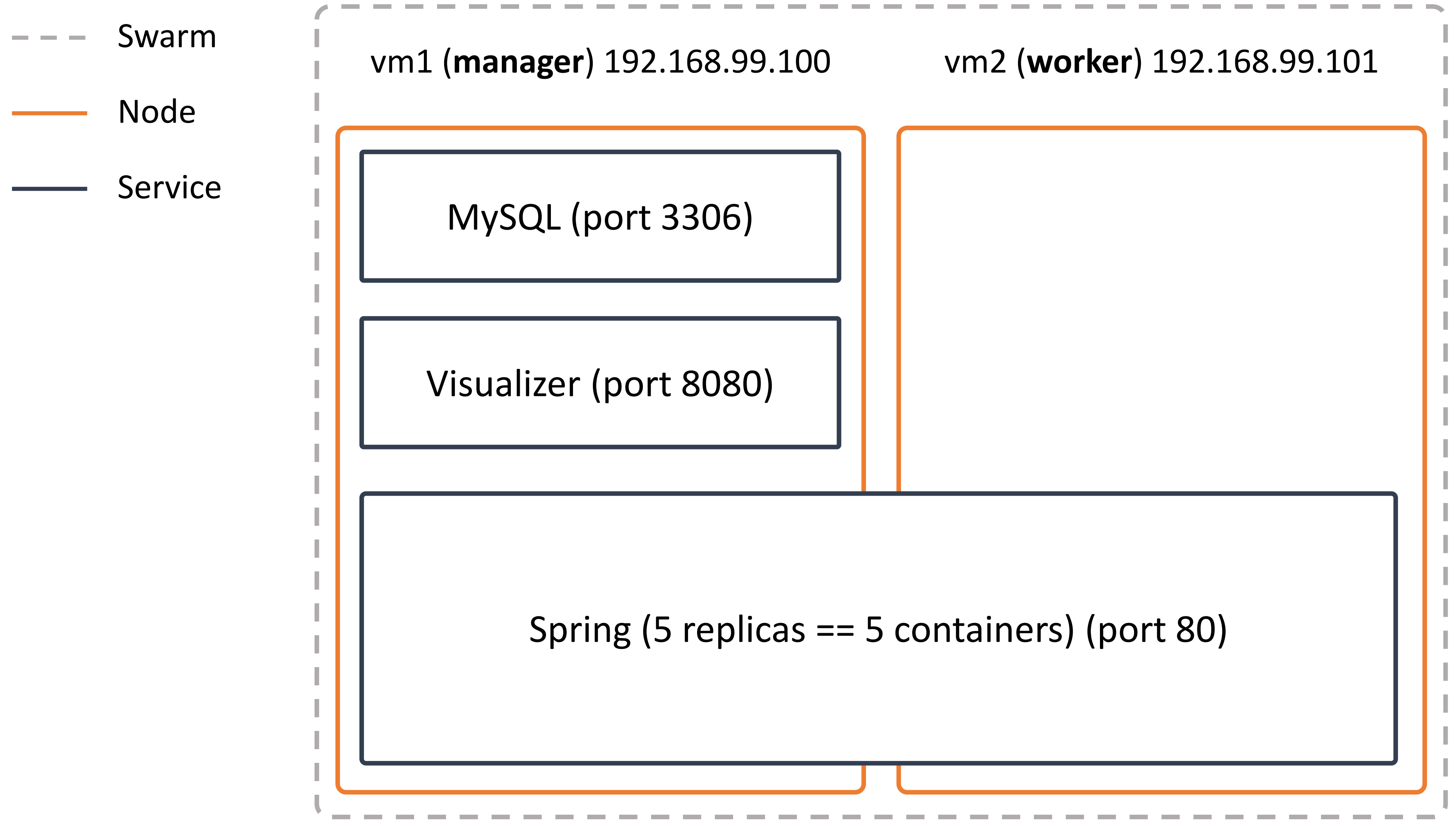
Application deployment

```
$ docker-machine ssh myvm1 "docker stack deploy -c docker-compose.yml librarywebapp"
```

Deployment - 2

```yaml
version: "3"
services:
  mysqldb:
    image: mysql:latest
    volumes:
      - my-datavolume:/var/lib/mysql
    deploy:
      placement:
        constraints: [node.role == manager]
    environment:
      - MYSQL_ROOT_PASSWORD=password
      - MYSQL_DATABASE=db_example
      - MYSQL_USER=springuser
      - MYSQL_PASSWORD=mysql2019
    networks:
      - webnet
  spring:
    image: davidmonnuar/springbootlibrarydemo:1.0
    depends_on:
      - mysqldb
    deploy:
      replicas: 5
      restart_policy:
        condition: on-failure
      resources:
        limits:
          cpus: "0.15"
          memory: 500M
    ports:
      - "80:8080"

    environment:
      - DATABASE_DRIVER=com.mysql.jdbc.Driver
      - DATABASE_HOST=mysqldb
      - DATABASE_PORT=3306
      - DATABASE_NAME=db_example
      - DATABASE_USER=springuser
      - DATABASE_PASSWORD=mysql2019
    networks:
      - webnet
  visualizer:
    image: dockersamples/visualizer:stable
    ports:
      - "8080:8080"
    volumes:
      - "/var/run/docker.sock:/var/run/docker.sock"
    deploy:
      placement:
        constraints: [node.role == manager]
    networks:
      - webnet
volumes:
  my-datavolume:
networks:
  webnet:
```

Deployment - 3


Swarm
Node
Service
vm1 (**manager**) 192.168.99.100
vm2 (**worker**) 192.168.99.101
MySQL (port 3306)
Visualizer (port 8080)
Spring (5 replicas == 5 containers) (port 80)

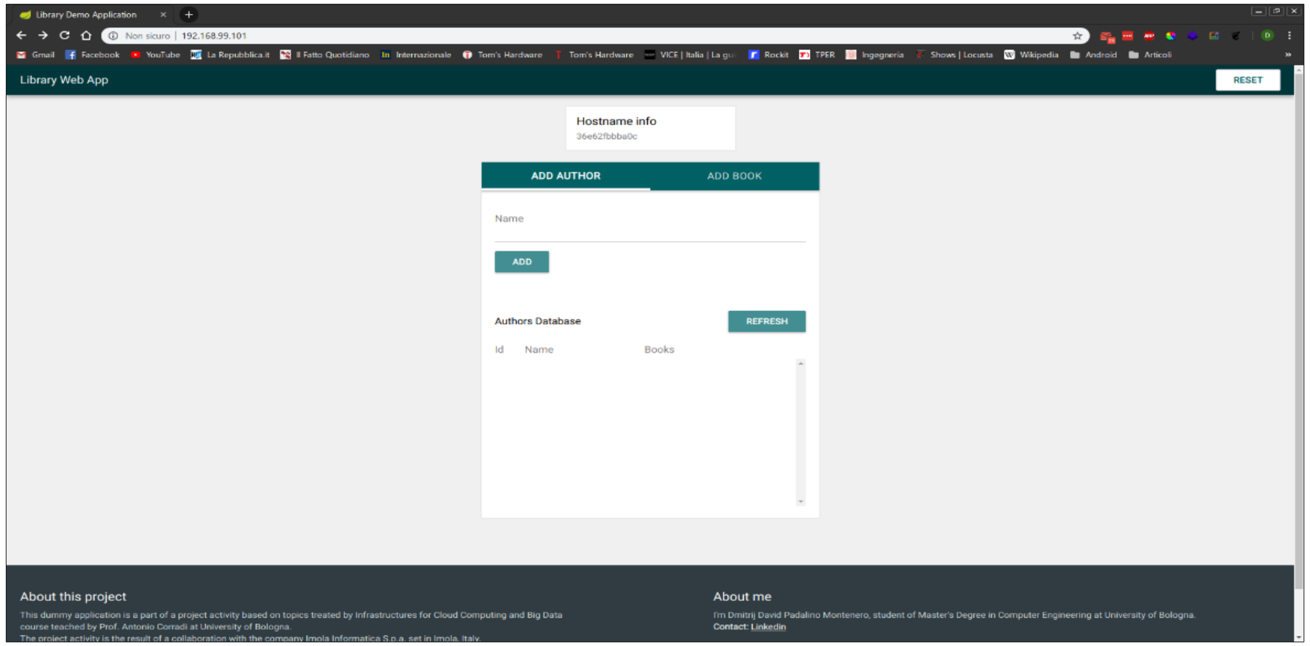
Deployment - 4

By inserting one of the two cluster node URLs in the browser it is possible to have **access to the application**

- http://192.168.99.100
- http://192.168.99.101

For the *Visualizer* service

- http://192.168.99.100 :8080
- http://192.168.99.101:8080



Library Web App

RESET

Hostname info
36e62fbbba0c

ADD AUTHOR ADD BOOK

Name

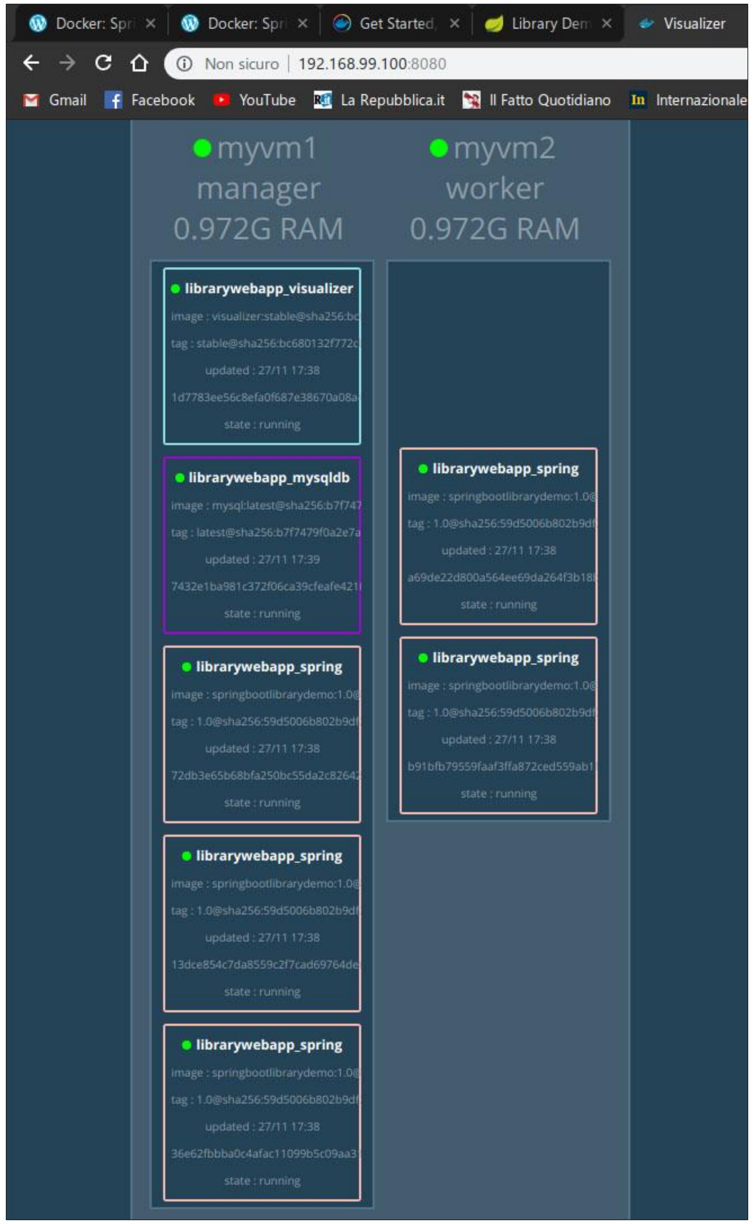
ADD

Authors Database REFRESH

Id Name Books

About this project
This dummy application is a part of a project activity based on topics treated by Infrastructures for Cloud Computing and Big Data course teached by Prof. Antonio Corradi at University of Bologna.
The project activity is the result of a collaboration with the company Imola Informatica S.p.a. set in Imola, Italy.

About me
I'm Dmitrij David Padalino Montenero, student of Master's Degree in Computer Engineering at University of Bologna.
Contact: Linkedin



192.168.99.100:8080

myvm1
manager
0.972G RAM

myvm2
worker
0.972G RAM

librarywebapp_visualizer
image : visualizer:stable@sha256:bc
tag : stable@sha256:bc680132f772c
updated : 27/11 17:38
1d7783ee56c8efa0f687e38670a08a
state : running

librarywebapp_mysqldb
image : mysql:latest@sha256:b7f747
tag : latest@sha256:b7f7479f0a2e7a
updated : 27/11 17:39
7432e1ba981c372f06ca39cfeafe421
state : running

librarywebapp_spring
image : springbootlibrarydemo:1.0@
tag : 1.0@sha256:59d5006b802b9df
updated : 27/11 17:38
72db3e65b68bfa250bc55da2c8264
state : running

librarywebapp_spring
image : springbootlibrarydemo:1.0@
tag : 1.0@sha256:59d5006b802b9df
updated : 27/11 17:38
13dce854c7da8559c2f7cad69764de
state : running

librarywebapp_spring
image : springbootlibrarydemo:1.0@
tag : 1.0@sha256:59d5006b802b9df
updated : 27/11 17:38
36e62fbbba0c4afac11099b5c09aa3
state : running

librarywebapp_spring
image : springbootlibrarydemo:1.0@
tag : 1.0@sha256:59d5006b802b9df
updated : 27/11 17:38
a69de22d800a564ee69da264f3b18
state : running

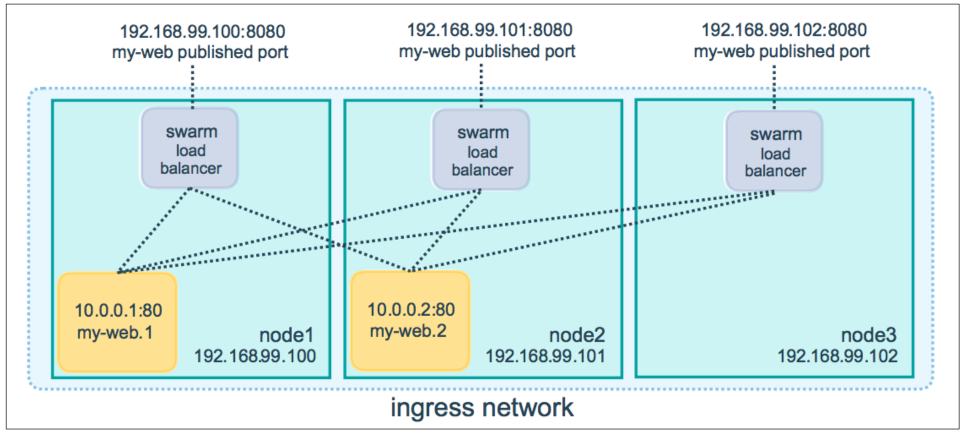
librarywebapp_spring
image : springbootlibrarydemo:1.0@
tag : 1.0@sha256:59d5006b802b9df
updated : 27/11 17:38
b91bfb79559faaf3ffa872ced559ab1
state : running

Deployment - 5 and Scaling

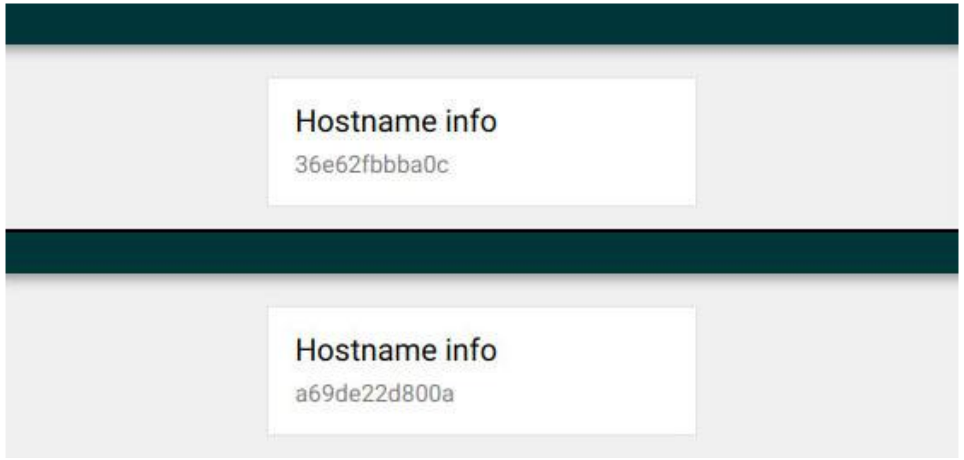
It is possible to access the app from the IP address of either myvm1 or myvm2

- http://192.168.99.100
- http://192.168.99.101

The reason both IP addresses work is that nodes in a swarm participate in an **ingress routing mesh**. This ensures that a service deployed at a certain port within your swarm always has that port reserved to itself, no matter what node is actually running the container.



The overlay network *webnet* is load-balanced by using a round-robin strategy.



Hostname info
36e62fbbba0c

Hostname info
a69de22d800a

Spring service horizontal scaling

You can scale the app by changing the replicas value in docker-compose.yml, saving the change, and re-running in the swarm manager

```
$ docker-machine ssh myvm1 "docker stack deploy -c
docker-compose.yml librarywebapp"
```

Docker recap

Docker is a powerful tool easy to use. The documentation available is very rich and it can be found here.

You’ve learned how to

- **containerize** an application by writing a Dockerfile of just few lines.
- **deploy** the app in a cluster
- **scale** the app

However to launch the application and scale it is necessary to access the cluster via ssh and by using Docker Swarm **is not possible** to define more fine-grained policies in order to dynamically scale the application.

From this experience it is interesting to understand how an orchestrator different from Docker Swarm, like **Kubernetes**, is able to tackle this problem.

Kubernetes

Kubernetes is an open-source container orchestration system designed by Google for automating application deployment, scaling and management inside a cluster.

The **main goal** is to hide the complexity of managing a fleet of containers by providing to the user a set of REST APIs.

Kubernetes is **portable** in nature, meaning it can run on various public or private cloud platforms such as AWS, Azure, OpenStack or Apache Mesos.

Features

kubernetes

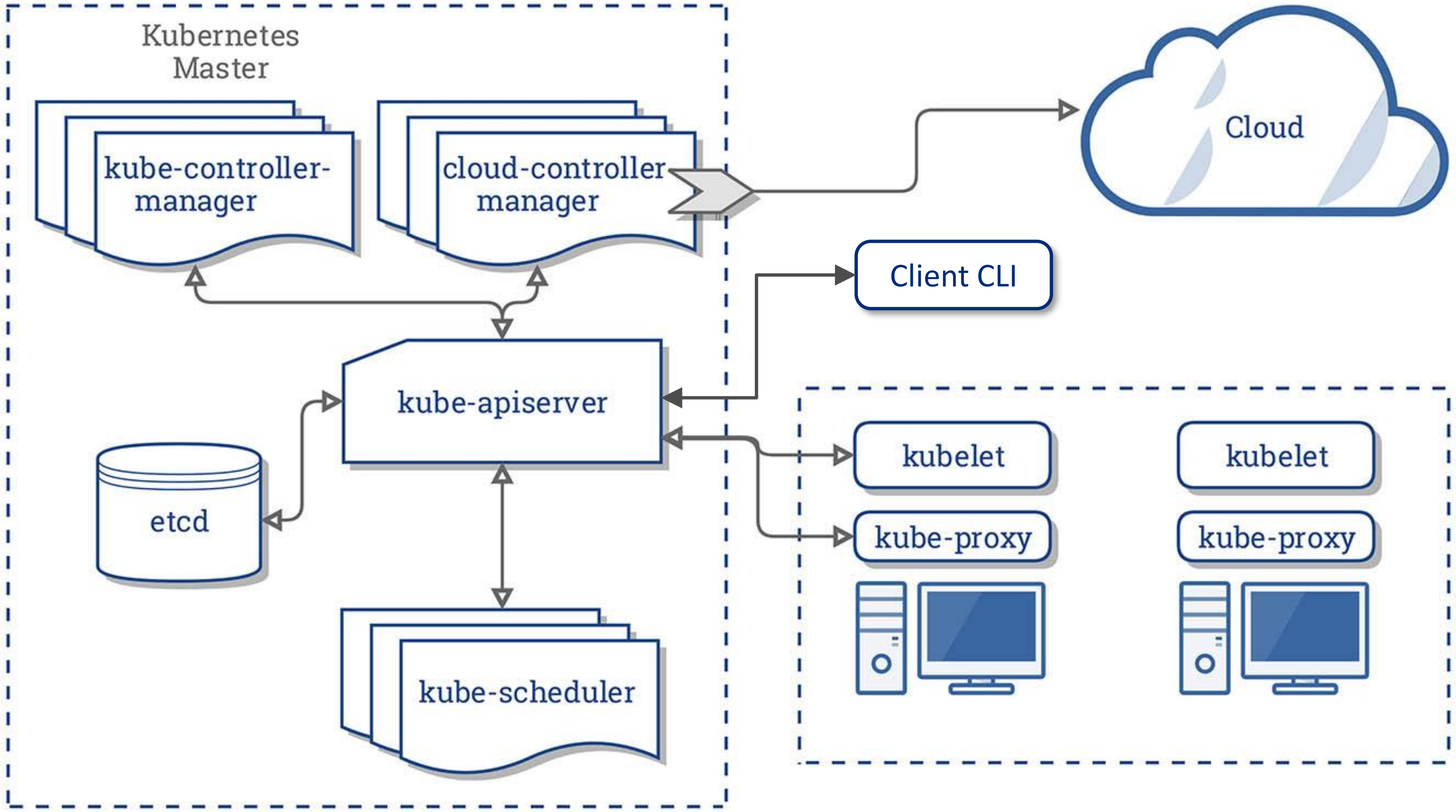
- Multi-container application deployment.
- Scale apps in a automatic and dynamic way.
- Roll out updates to your app or its configuration by preserving service availability.
- It provides computational, network and storage resources to your app and a discovery service.
- Independency from the underlying infrastructure.

Kubernetes is more powerful than Docker Swarm, and requires more work to deploy.

However the work is intended to provide a big payoff in the long run in terms of a more manageable, resilient application infrastructure

Architecture

kubernetes



Master-slave architecture.

Master node

- Etcd
- Kube-apiserver
- Kube-controller-manager
- Kube-scheduler

Worker node

- Kubelet
- Kube-proxy

Client

- CLI
- Dashboard

Basic concepts

- Pod
  represents a single istance of an application or running process in Kubernetes, and consists of one or more containers. Kubernetes starts, stops, and replicates all containers in a pod as a group.
- Service
  because pods live and die as needed, we need a different abstraction for dealing with the application lifecycle. An application is supposed to be a persistent entity, even when the pods running the containers that comprise the application aren't themselves persistent. To that end, Kubernetes provides an abstraction called a service. A service describes how a given group of pods (or other Kubernetes objects) can be accessed via the network.
- Volume
  similar to Docker Volumes, the main difference in Kubernetes is that the association between a volume and the container is done at the pod level. So the volume is attached to all containers running in that pod.
- Namespace
  a virtual cluster (a single physical cluster can run multiple virtual ones) intended for environments with many users spread across multiple teams or projects, for isolation of concerns. Also, a namespace can be allocated a resource quota to avoid consuming more than its share of the physical cluster's overall resources.

Kubernetes Lab

Starting from the same web app (Spring and MySQL)

- Cluster creation and configuration with Minikube
- Deployment
- Static and dynamic horizontal scaling

CPU	Intel(R) Core(TM) i7-3610QM CPU @ 2.30Ghz
RAM	8GB
SSD	Samsung 850 Evo 500GB
Interfaccia di rete wireless	Qualcomm Atheros AR5BWB22
Sistema operativo	Ubuntu 18.04.1 LTS Bionic Beaver
Versione Oracle VM VirtualBox	5.2.20r125813
Versione Minikube	0.30.0
Versione kubectl Client	1.12.3
Versione kubectl Server	1.10.0
Versione Google Chrome	69.0.3497.100 a 64 bit

Cluster creation and configuration - 1

kubernetes

Minikube is a tool that makes it easy to run Kubernetes locally.

It runs a single-node Kubernetes cluster inside a VM.

Cluster initialization

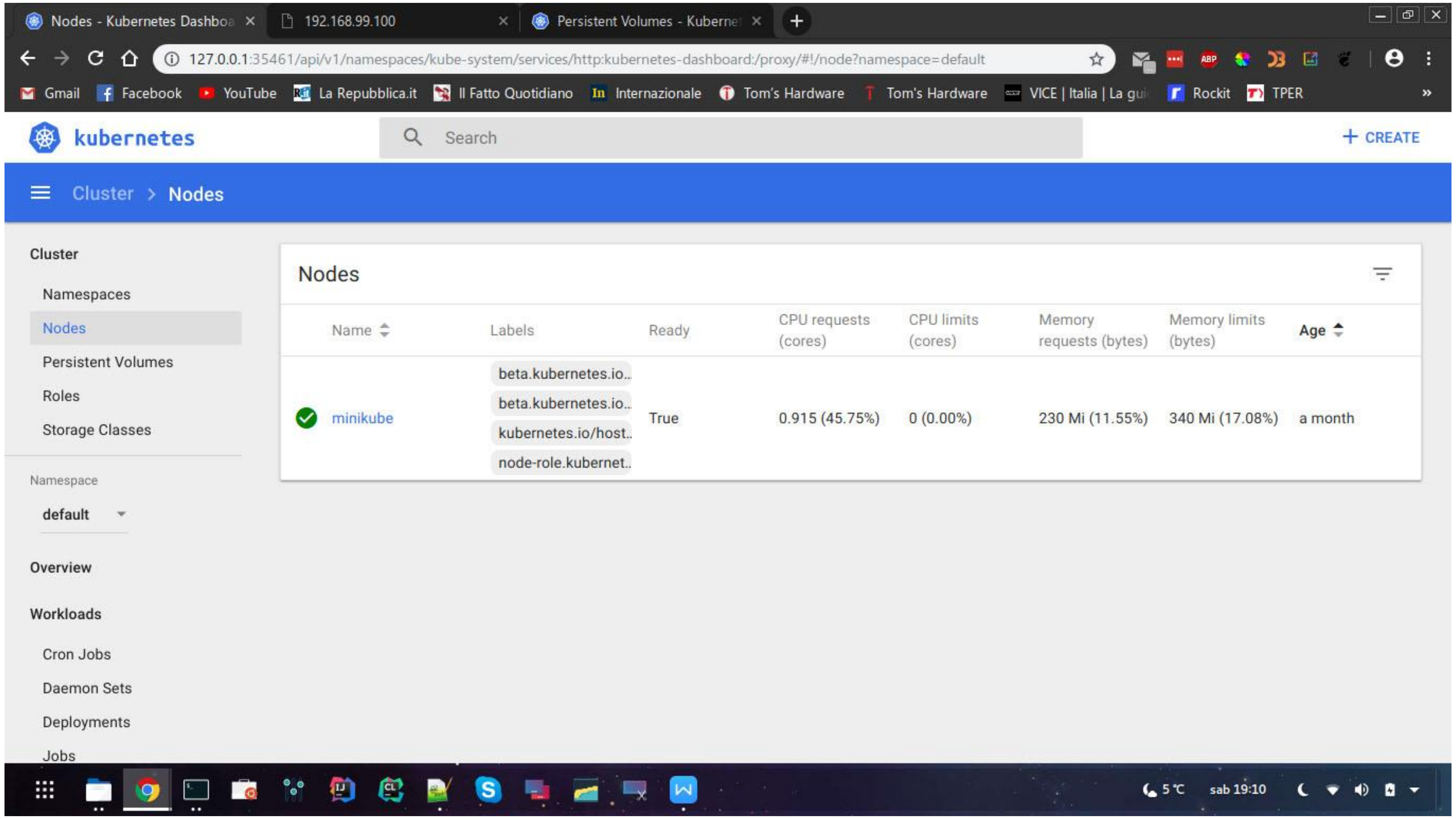
```
$ minikube start

$ kubectl cluster-info
Kubernetes master is running at https://192.168.99.100:8443
CoreDNS is running at https://192.168.99.100:8443/api/v1/namespaces/kubesystem/services/kube-dns:dns/proxy

$ kubectl get nodes
NAME         STATUS         ROLES        AGE         VERSION
minikube     Ready          master       40d         v.1.10.0
```

Launch Dashboard

```
$ minikube dashboard
```

Cluster creation and configuration - 2

Dashboard



Nodes

Name	Labels	Ready	CPU requests (cores)	CPU limits (cores)	Memory requests (bytes)	Memory limits (bytes)	Age
✓ minikube	beta.kubernetes.io.. beta.kubernetes.io.. kubernetes.io/host.. node-role.kubernet..	True	0.915 (45.75%)	0 (0.00%)	230 Mi (11.55%)	340 Mi (17.08%)	a month

Deployment - 1

App organization

- One pod for Spring.
- One pod MySQL + Volume.
- One service for MySQL to connect the MySQL pod to the Spring client pod.
- One service for Spring to expose publicly the whole app.

Two separated pods in order to scale only the Spring pod.

The first step is to write for each app component the related **Deployment file**.

Deployment - 2

kubernetes

spring-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: spring
  labels:
    app: spring
  spec:
    replicas: 1
    selector:
      matchLabels:
        app: spring
    template:
      metadata:
        labels:
          app: spring
      spec:
        containers:
        - name: spring
          image: davidmonnuar/springbootlibrarydemo:1.0
          env:
          - name: DATABASE_DRIVER
            value: com.mysql.jdbc.Driver
          - name: DATABASE_HOST
            value: mysqldb
          - name: DATABASE_PORT
            value: "3306"
          - name: DATABASE_NAME
            value: db_example
          - name: DATABASE_USER
            value: springuser
          - name: DATABASE_PASSWORD
            value: mysql2019
          ports:
          - containerPort: 8080
```

spring-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-service
  labels:
    run: web-service
spec:
  type: NodePort
  ports:
  - port: 8080
    protocol: TCP
  selector:
  app: spring
```

Deployment - 3

mysql-deployment.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mysqldb
spec:
  ports:
  - port: 3306
  selector:
    app: mysql
  clusterIP: None
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mysql
spec:
  selector:
    matchLabels:
      app: mysql
  strategy:
    type: Recreate
  template:
    metadata:
      labels:
        app: mysql
    spec:
      containers:
      - image: mysql:5.7
        name: mysql
        env:
          # Use secret in real usage
        - name: MYSQL_ROOT_PASSWORD
          value: password
        - name: MYSQL_DATABASE
          value: db_example
        - name: MYSQL_USER
          value: springuser
        - name: MYSQL_PASSWORD
          value: mysql2019
        ports:
        - containerPort: 3306
          name: mysql
        volumeMounts:
        - name: mysql-persistent-storage
          mountPath: /var/lib/mysql
        volumes:
        - name: mysql-persistent-storage
          persistentVolumeClaim:
            claimName: mysql-pv-claim
```

Deployment - 4

kubernetes

mysql-pv.yaml

```yaml
kind: PersistentVolume
apiVersion: v1
metadata:
  name: mysql-pv-volume
  labels:
    type: local
  spec:
    storageClassName: manual
    capacity:
      storage: 1Gi
    accessModes:
      - ReadWriteOnce
    hostPath:
      path: "/mnt/data"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
spec:
  storageClassName: manual
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

Deployment (minikubeSetup.sh)

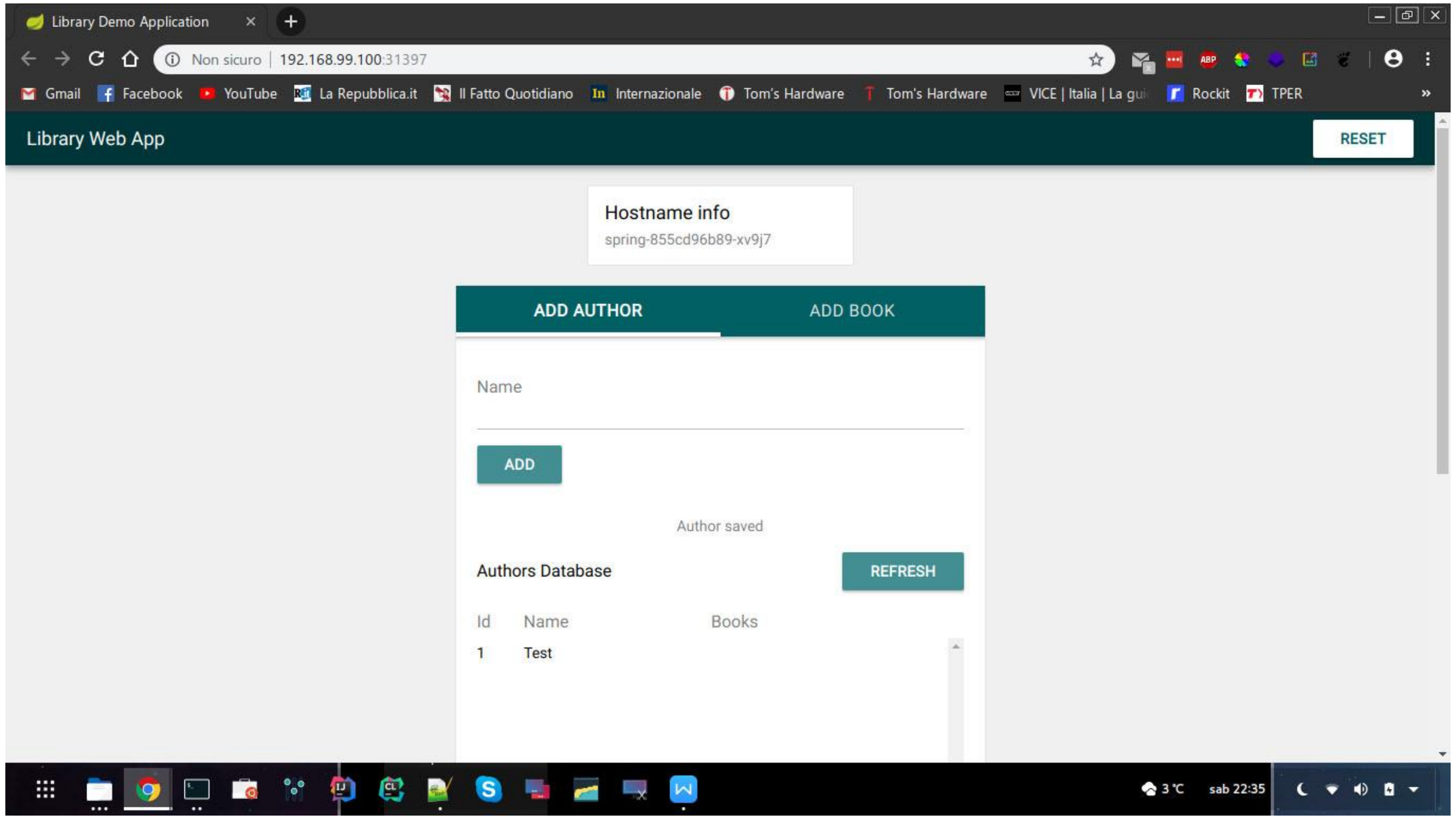
```sh
#!/bin/sh
kubectl create -f ./mysql/mysql-pv.yaml
kubectl create -f ./mysql/mysql-deployment.yaml
kubectl create -f ./spring/spring-deployment.yaml
kubectl create -f ./spring/spring-service.yaml
kubectl get services
minikube ip
```

```
$ ./minikubeSetup.sh
persistentvolume/mysql-pv-volume created
persistentvolumeclaim/mysql-pv-claim created
service/mysqldb created
deployment.apps/mysql created
deployment.apps/spring created
service/web-service created
NAME         TYPE        CLUSTER_IP     EXTERNAL-IP    PORT(S)          AGE
kubernetes   ClusterIP   10.96.0.1      <none>         443/TCP          47m
mysqldb      ClusterIP   None           <none>         3306/TCP         1s
web-service  NodePort    10.96.240.147  <none>         8080:31397/TCP   0s
192.168.99.100
```

Deployment - 4

To access the deployed app

```
$ minikube service web-service
```



Hostname info
spring-855cd96b89-xv9j7

Author saved

Authors Database

Id	Name	Books
1	Test	

The string reported in the *Hostname Info* box is the pod id in which the container is running.

In Docker that string was the container id.

Spring pod scaling

kubernetes

Static horizontal scaling by using CLI

```
$ kubectl scale deployments/spring --replicas=2

$ kubectl get pods
NAME                       READY   STATUS     RESTARTS    AGE
mysql-7c9fc47c5d-z892k     1/1     Running    0           2h
spring-855cd96b89-2hmwh    1/1     Running    0           2h
spring-855cd96b89-xv9j7    1/1     Running    0           2h
```

Dynamic horizontal scaling by defining a scaling policy

- Addon Minikube **metric-server**
- Kuberntes object **Horizontal Pod Autoscaler** (HPA)
- Some changes needed for the spring deployment file

```
…
  template:
    metadata:
      labels:
        app: spring
    spec:
      containers:
      - name: spring
        image: davidmonnuar/springbootlibrarydemo:1.0
        resources:
          requests:
            cpu: "500m"
            limits:
              cpu: "0.5"
        env:
        - name: DATABASE_DRIVER
…
```

As scaling metric we focus on CPU resources usage

```
$ minikube addons enable metrics-server

$ kubectl autoscale deployment spring --cpu-percent=50 --min=1 --
max=10

$ kubectl run -i --tty load-generator --image=busybox /bin/sh
#/ while true; do wget -q -O- http://192.168.99.100:31580/; done
```

Pod replicas before the workload

```
$ kubectl get hpa
NAME      REFERENCE           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
spring    Deployment/spring   0%/50%    1         10        1          1m
```

Pod replicas 1 minute after the workload start

```
$ kubectl get hpa
NAME      REFERENCE           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
spring    Deployment/spring   31%/50%   1         10        1          2m
```

Pod replicas 3 minutes after the workload start

```
$ kubectl get hpa
NAME      REFERENCE           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
spring    Deployment/spring   30%/50%   1         10        2          4m
```

Pod replicas after the workload end

```
$ kubectl get hpa
NAME      REFERENCE           TARGETS   MINPODS   MAXPODS   REPLICAS   AGE
spring    Deployment/spring   0%/50%    1         10        1          9m
```

Kubernetes Recap

Kubernetes is a **powerful** and **effective** tool for containerized apps management.

It is significantly **more complex than Docker Swarm** however it is more flexible.

The **high-level abstractions** that Kubernetes provides to deploy an application let a definition of fine-grained policies and as a consequence the app management is simplified.

Kubernetes is much more than what we’ve seen so far.

The documentation can be found here.

OpenShift

OpenShift is a family of containerization software developed by Red Hat. Its flagship product is the OpenShift Container Platform (OpenShift from now on), a PaaS for private clouds built around **Docker containers** orchestrated and managed by **Kubernetes** on a foundation of **Red Hat Enterprise Linux**.

Developement-oriented new features

- Deployment from the Dockerfile.
- Deployment of pre-built Docker images.
- **Deployment directly from the source code** (GitHub).

A **constraint** is that in order to use OpenShift you need to use Red Hat Enterprise Linux or Red Hat Atomic.

Kubernetes extensions

- Several new objects
- Project
- Route

Minishift is a tool that helps you run OpenShift locally by running a single-node OpenShift cluster inside a VM. You can try out OpenShift or develop with it, day-to-day, on your local host. Link here.

The **main advantage** of this plaform is to be closer to software developers than Kubernetes.

IBM Cloud

IBM Cloud (formerly IBM Bluemix) is a suite of cloud computing services that offers both PaaS and Infrastructure as a Service (IaaS). Among the huge amount of services that IBM Cloud makes available there is also IBM Cloud Kubernetes Service.

In order to use the **Kubernetes-as-a-Service** you need to create a trial account on IBM Cloud and in few seconds you have the access to a single node Kubernetes cluster.

The Kubernetes service is very close to the vanilla Kubernetes provided by Google and it does not contain strong customizations like OpenShift.

Surely the **main advantage** of IBM Cloud is that it quickly provides a Kubernetes cluster off-the-shelf and ready to use without spend too much time in cluster configuration.

Latest news: IBM acquired Red Hat.

Conclusions

Docker and **Kubernetes** have completely re-inveted the way the cloud industry faces the software development, deployment and maintenance.

The idea of exploiting the native capability of operating systems for isolating the resources by providing the support for containers is successful.

The **containerization** has proven to be an innovative technology that has become in a short time, given the growing adoption rate, a **crucial tool in the modern cloud solutions**.

Source code link: https://github.com/davidMonnuar/projectActivity

Thank you for your attention

Containerization workshop

Infrastructures for Cloud Computing and Big Data M

Dmitrij David Padalino Montenero

Academic Year 2018/2019