Containers & Orchestration

Release Engineering for Machine Learning Applications
(REMLA, CS4295)

Sebastian Proksch
S.Proksch@tudelft.nl

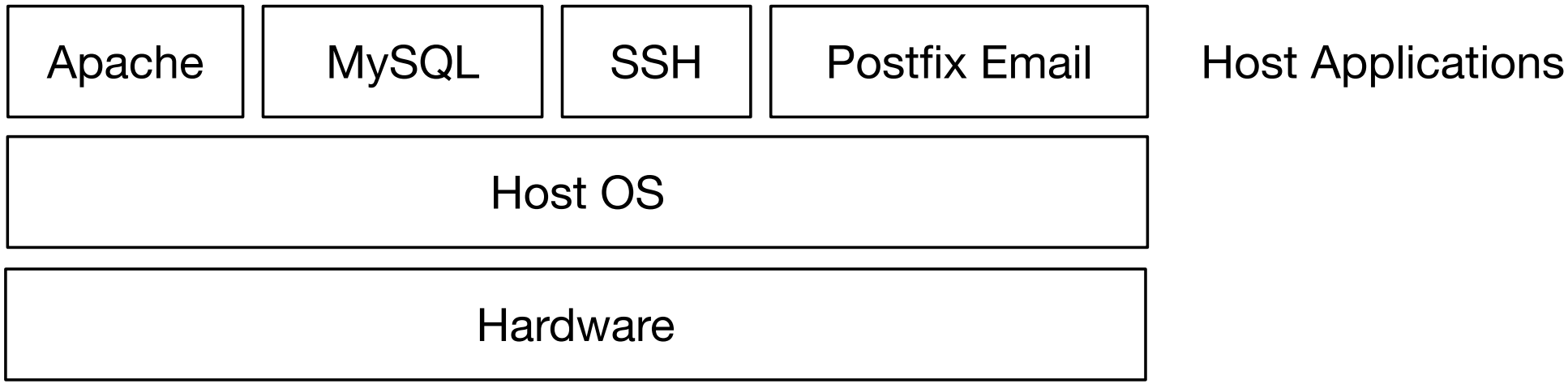
Luís Cruz
L.Cruz@tudelft.nl

Goal of today…

- Know different abstraction styles of server hosting
- Understand challenges when hosting services
- Understand basic concepts of Docker and know how to create and run your own image
- You can run single-host deployments of distributed systems using Docker Compose
- You know the high-level concepts and terminology of Kubernetes

(Service) Hosting

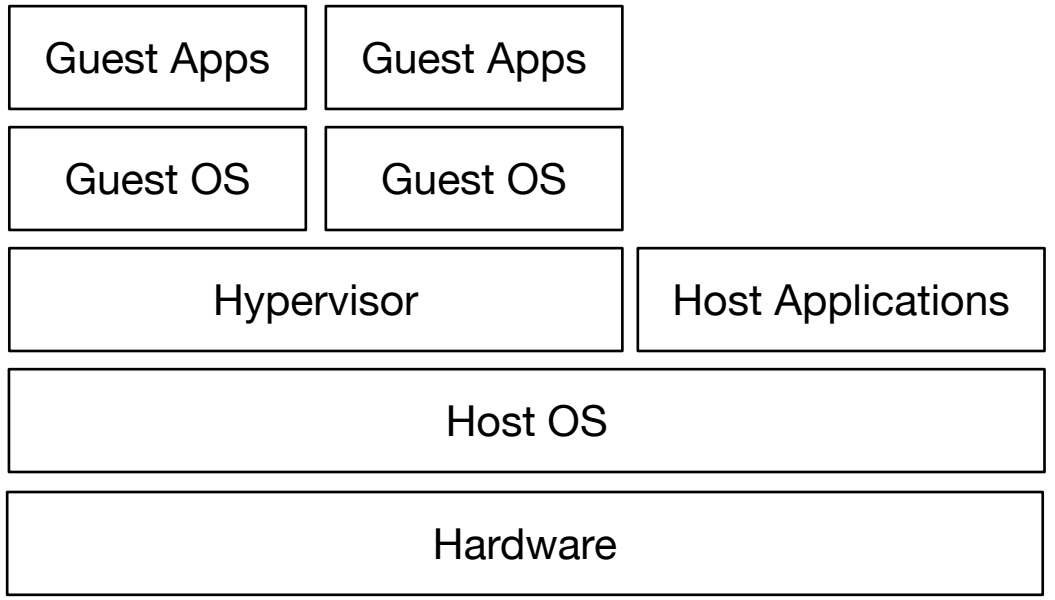
The Old Ways of the Sysadmins



- All applications running in the same environment
- Abstraction/isolation through multiple hosts
- Security through user/group permissions

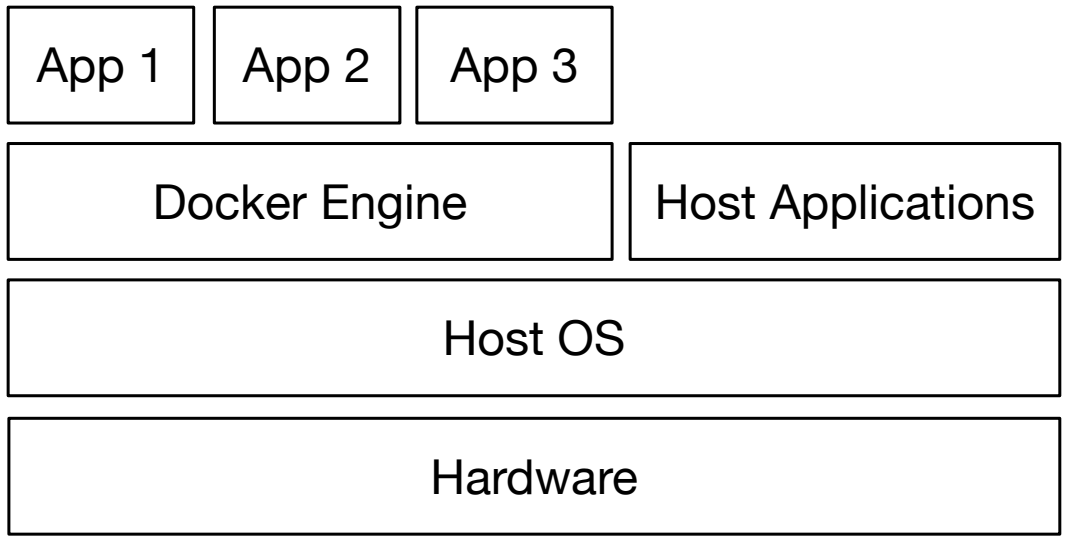
Virtual Machines

- Hypervisor introduces virtual hardware
- High Flexibility for Guests
- Overhead at each level
- Resources are usually reserved at start-up
- “Normal” boot times



Containers

- No separate OS needed
- Containers use same kernel
- Containers run as processes
- “Minimal” resource reqs
  - Memory
  - Disk (No redundancy)
- Almost instant boot



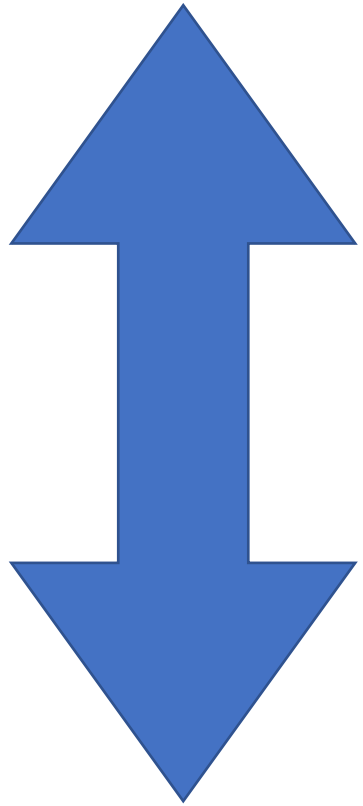
Challenges

Infrastructure

- Conflicting requirements for hosting infrastructure
  (Java 6 for Service 1, Java 11 for Service 2, ...)
- Maintenance Cost
  (effort for update/configuring/optimizing servers)
- Provisioning Effort
  (compare setting up bare-metal with a VM)

Scaling

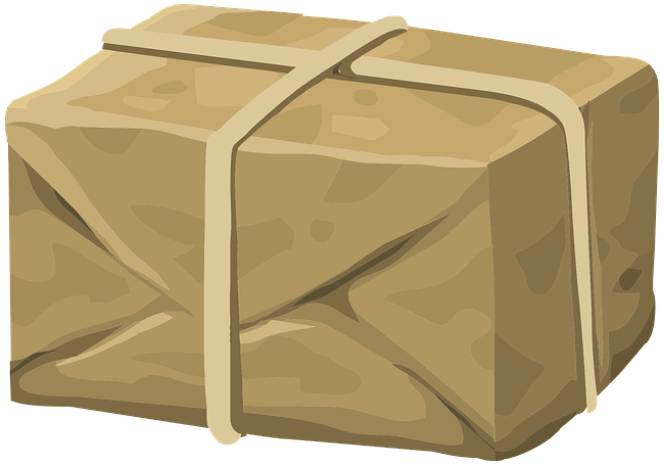
- Vertical scaling (powerful machines)
- Horizontal scaling (more machines)



- How to distribute load?
- Elasticity (Christmas Effect)
- Avoid redundancy

Packaging & Distribution

- Libraries vs. Applications
  - Intra- vs. Extra-Ecosystem
  - Runtime Environment
- Configuration Management
- Required Operation Data



Portability

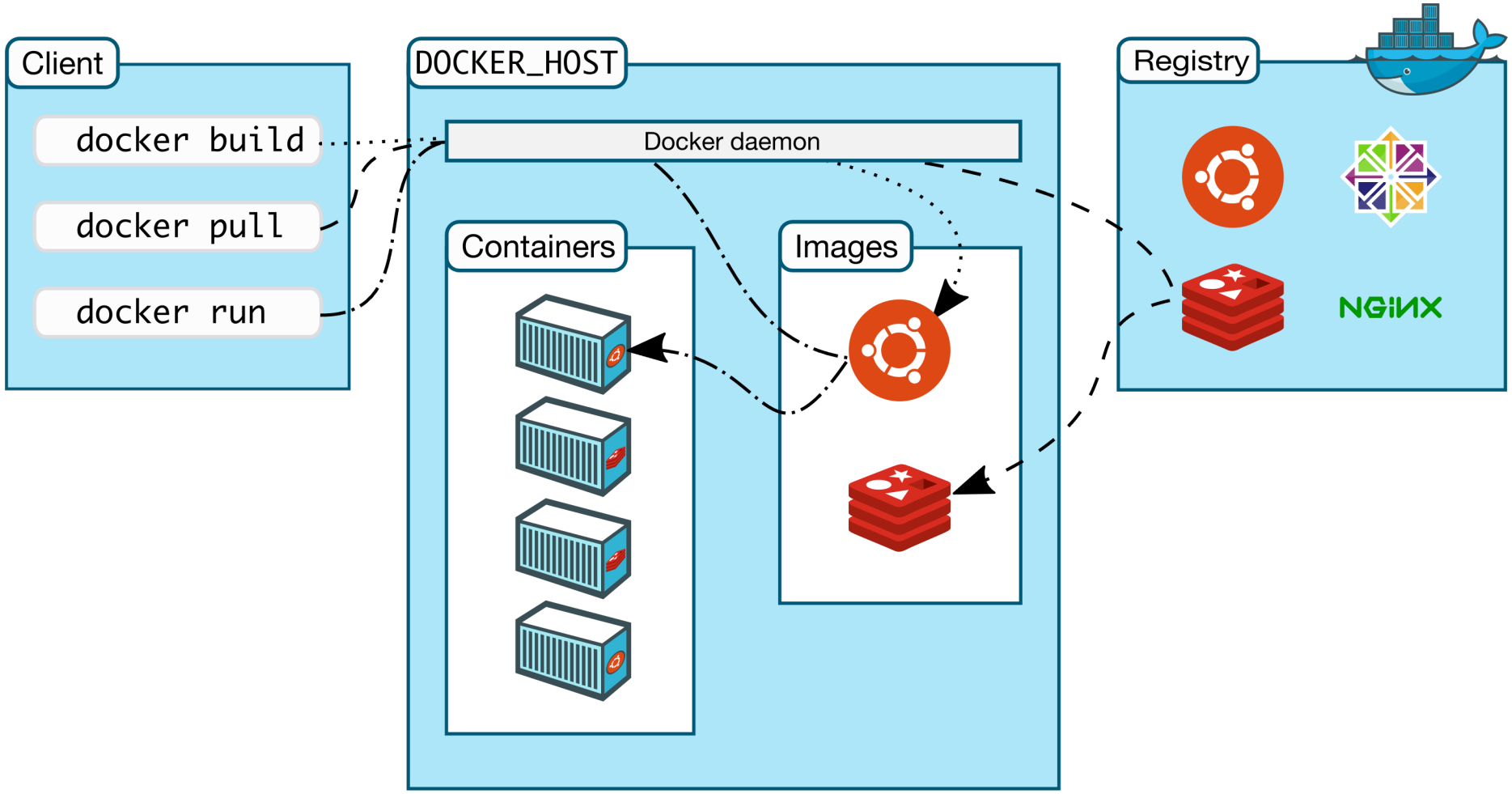
- “Works on my machine” problems
- Make software independent of OS or env.
- Allow moving between machines
  (Future updates, migrations)
- Development, Integration Testing, Production
- Replicability

Security

- Online systems will be attacked
- Make updates easy
- Prepare for the worst and expect breaches
- Intruders should not be able to gain full control
  - Protect data (read, but also alter)
  - Protect resources (computing power, network)
- Reliable monitoring and logging

Docker

System Architecture



Source: Docker

Isolated Execution

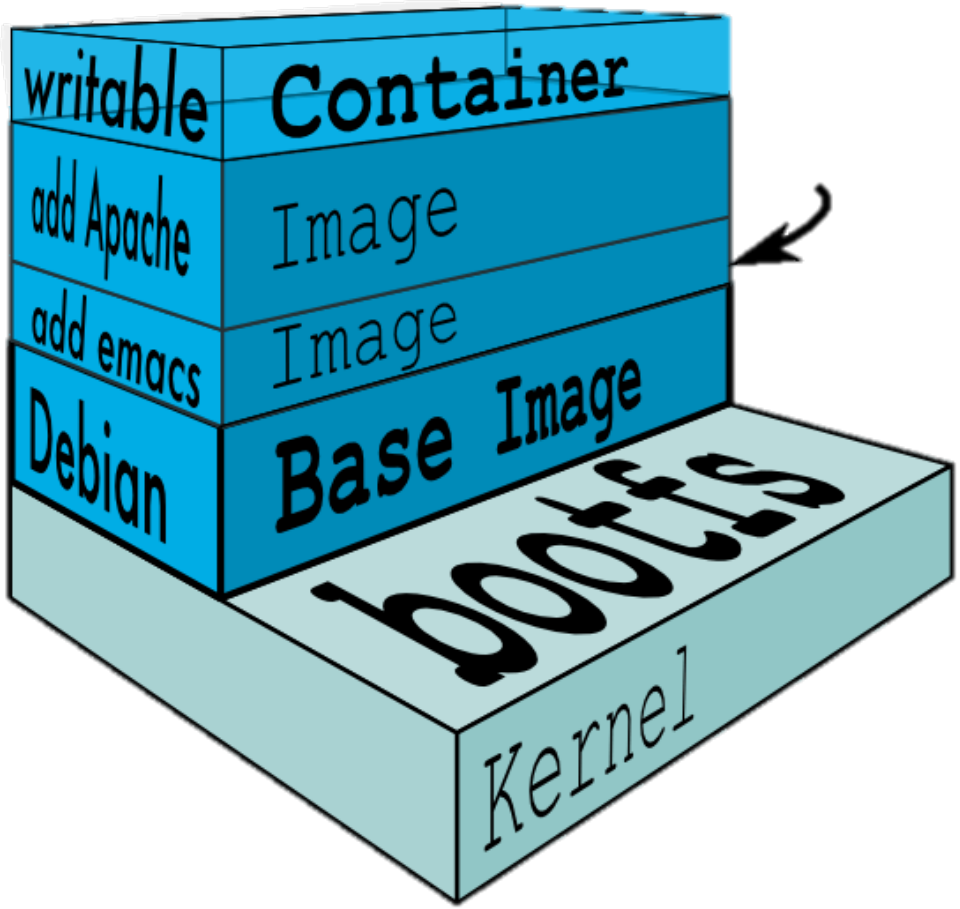
- Run as process, not as virtual machine
- Leverages established Linux concepts
  - Control Groups (limit resources for a process)
  - Namespaces (limit access for a process)
  - Seccomp (limit usable kernel features)

Networking

- Network access of containers is managed
- No network for container (None)
- Container uses host network (Host)
- Fake “physical network device” (Macvlan)
- Default: Virtual network routed through host (Bridge)

Union File System

- Containers are instances of Images
- Each layer is a diff to parent
- Images are read-only
- Containers are writable
- Avoids redundancy
- One Image can be used multiple times (tree!)



Source: Docker

Volumes

- Volumes can be mounted into container at runtime
  - bind paths on host machine (More Convenient)
  - In-memory mounts (If no persistence is required)
  - Mount block-level files (More Performant)
- Use Volumes to...
  - ... persist information
  - ... avoid growing containers (e.g., logging!)
  - ... share data between containers

Break?

How to create a Docker image?

Build an Image Through a Dockerfile

Dockerfile

```
FROM openjdk:11.0-jre-slim
ENTRYPOINT [”bash"]
```

Base Image

Define entry point that dispatches commands

```
$ docker build .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM openjdk:11.0-jre-slim
 ---> 973c18dbf567
Step 2/2 : ENTRYPOINT ["bash"]
 ---> Using cache
 ---> e411903763a6
Successfully built e411903763a6
$
```

Every individual step of Dockerfile generates new image

Images are identified by digests/hashes

```
$ docker run -it e411903763a6
root@3300a82e5125:/#
```

Images can be **`run`** (-i: interactive, -t: terminal)

Images Can Be Named (tagged)

```
$ docker build -t a -t b/c -t d:1.2.3 -t e/f/g/h/i .
Sending build context to Docker daemon  2.048kB
Step 1/2 : FROM openjdk:11.0-jre-slim
 ---> 973c18dbf567
Step 2/2 : ENTRYPOINT ["bash"]
 ---> Using cache
 ---> e411903763a6
Successfully built e411903763a6
Successfully tagged a:latest
Successfully tagged b/c:latest
Successfully tagged d:1.2.3
Successfully tagged e/f/g/h/i:latest
$
```

Multiple tags can be used at once

Tags resolve to digest

Best practice would be to publish:

- user/repo:1.2.3
- user/repo:1.2
- user/repo:1
- user/repo:latest

Distribution via Container Registry

- Images uniquely represented with digest
- Human identification of an image
  - User/Organization (empty for “official images”)
  - Repository
  - Tag (can be name, version, meta data)
  - OS/Arch
- Use via docker pull / docker push

Default registry is dockerhub.com, but can easily changed.

ENTRYPOINT versus RUN

```
FROM openjdk:11.0-jre-slim
ENTRYPOINT [”java"]
CMD [“-version”]
```

```
docker build –t remla .
```

Custom ENTRYPOINT

```
$ docker run -it remla
openjdk version "11.0.7" 2020-04-14
...
$
```

```
$ docker run -it --entrypoint ls remla
bin dev home lib64 mnt proc run srv tmp var
boot etc lib media opt root sbin sys usr
$
```

Custom RUN

```
$ docker run -it remla --help
Usage: java [options] <mainclass> [args...]
...
$
```

```
$ docker run -it --entrypoint ls remla -l
total 64
drwxr-xr-x   2 root root 4096 May 14  2020 bin
...
$
```

Preparing the Image Content

```
FROM ubuntu:latest

WORKDIR /root/
COPY somefile.txt .
RUN apt update
RUN apt install wget

ENTRYPOINT ["bash"]
```

The **WORKDIR** defines the current working directory in the image.

You can **COPY** or **ADD** existing files/folders into to the image. **ADD** has a lot of additional features (e.g., auto extracting archives), which makes it very unpredictable. For a more transparent execution, use the (more limited) **COPY**.

You can freely change the system through **RUN** commands.

Multi-Stage Builds

```
FROM openjdk:11.0-jre-slim AS first

RUN java --help > help.txt
RUN java --version > version.txt

FROM ubuntu:latest

WORKDIR /root/
RUN mkdir data
COPY --from=first version.txt .

ENTRYPOINT ["bash"]
```

Avoid unnecessary image grow (e.g., `apt-get update`)

Use separate stage to run expensive command (e.g., compilation), `COPY` relevant output (e.g., just the binary)

Stages can be named, by default, they are numbered.

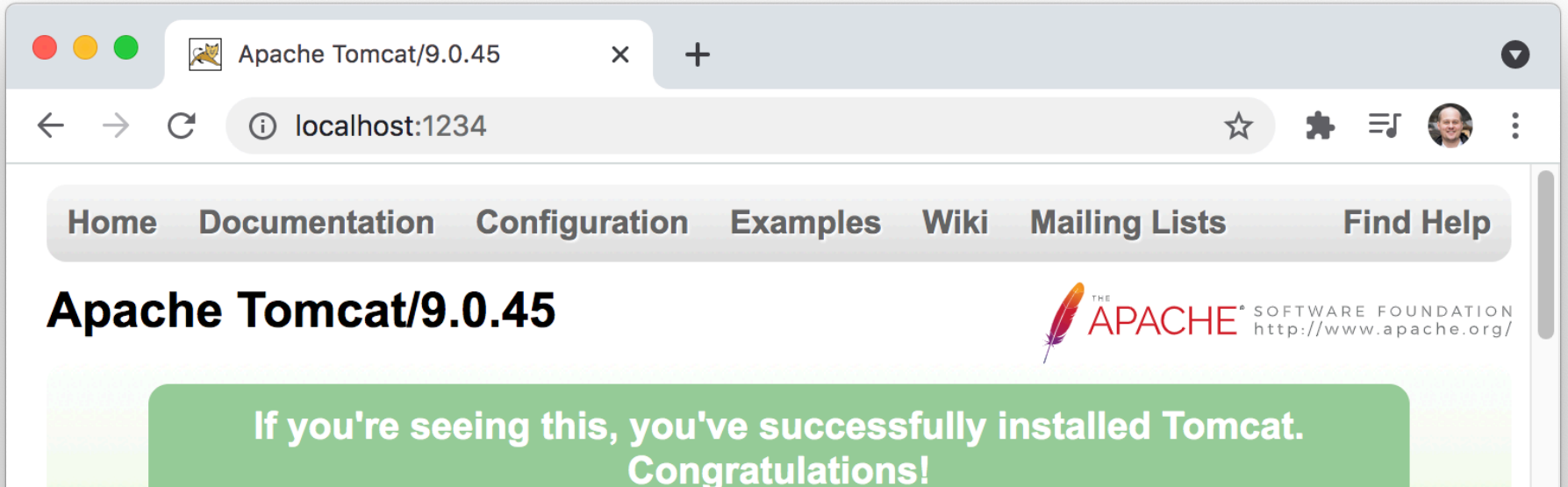
Offering Network Services

```
...
EXPOSE 8080
...
```

Images that want to offer network services must **EXPOSE** the port, at which the service is running.

To make this port available, the `docker run` command must map the port to a local port.

```
docker run –d –p 1234:8080 tomcat
```



Docker client has many more peculiarities…

- Mounting volumes
- Life-cycle Management (start/stop/remove)
- Image management
- Connecting to running containers
- Garbage collection
- … **We will cover more details in the tutorial.**

Orchestration

Docker Compose: Single-Host Deployment

```yaml
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "5000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    links:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

Define Services, register DNS Records

Build local Docker image

Open Ports to Host

Mount Volumes

Connect Containers

Networking

```
docker-compose up -d
```

Advanced Docker Compose

- Environment Variables
- Custom RUN commands
- Advanced Networking
- Startup Management (depends_on)
- Inheritance
- … https://docs.docker.com/compose/

Cluster Management

Docker Swarm
Kubernetes

- Abstraction for Multi-host Management
  (aka. Cluster/Cloud/Farm/...)
- Desired State Reconciliation
- Service Discovery
- Load-Balancing
- Auto-Scaling

Kubernetes

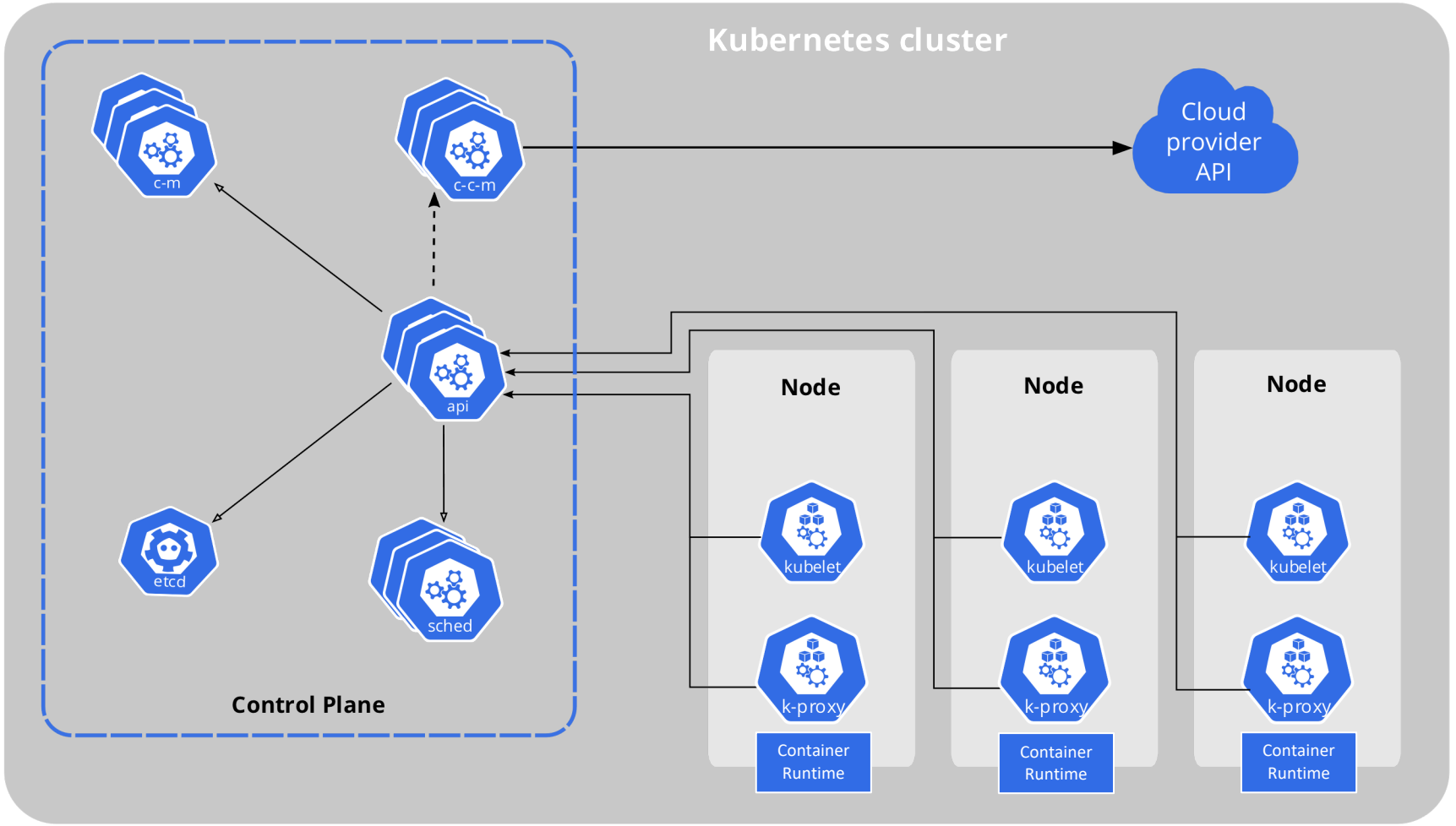
Disclaimer: This lecture can only give a first glimpse into the capabilities of Kubernetes. The tutorial will put this into a more practical context, but there is no way to cover all or even just a representative subset of all Kubernetes features!

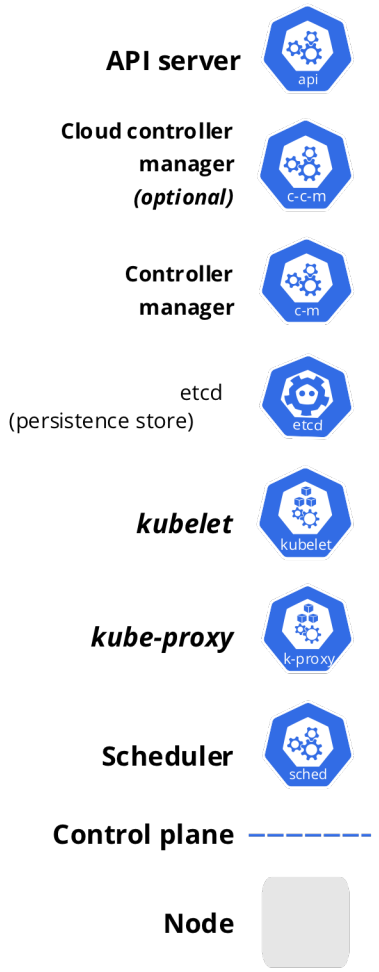
Why you need Kubernetes and what it can do

- **Service discovery and load balancing** Kubernetes can expose a container using DNS or IP address. On high traffic, Kubernetes can load-balance and auto-scale.
- **Storage orchestration** Kubernetes allows you to automatically mount a storage system of your choice, such as local storages, public cloud providers, and more.
- **Automated rollouts and rollbacks** Kubernetes will keep the actual state in sync with the configured desired state, at a controlled rate.
- **Automatic bin packing** Kubernetes will distribute required containers within the cluster of nodes to optimize resource utilization.
- **Self-healing** Kubernetes restarts failed containers, replaces containers, kills irresponsive containers, and only advertise functional clients.
- **Secret and configuration management** Store and manage sensitive information. Deploy and update secrets without rebuilding your container images or exposing secrets in your configuration.

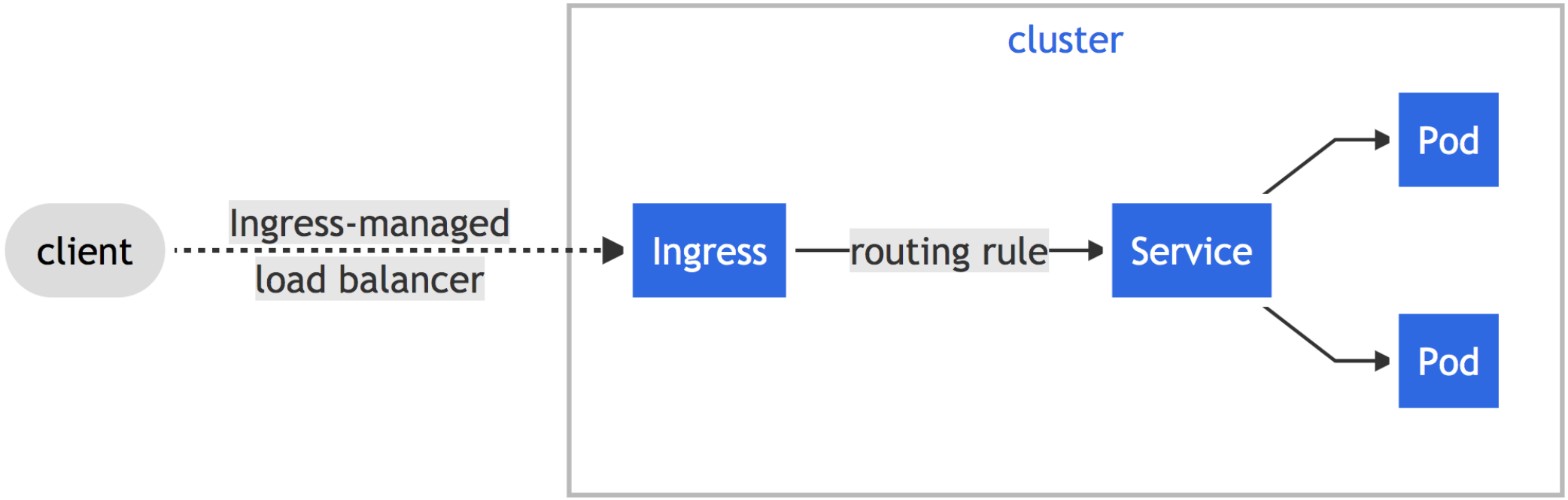
https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Architecture





Ingress



- Ingress/Ingress Controller Makes Service Available
- Single entry/exit point of the cluster

Service

- An abstract way to expose an application running on a set of Pods as a network service.
- In Kubernetes, DNS is used for service discovery.
- Pods get their own IP addresses. Grouping multiple pods with a single DNS name allows load-balancing.

Pod (as in a pod of whales or pea pod)

- Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.
- A Pod is a group of one or more containers, with shared storage and network resources.
- A Pod's contents are always co-located and co-scheduled, and run in a shared context.
- A Pod models a "logical host” and tightly coupled containers, very often just one.

Deployment

- A Deployment is a declaration of a desired state.
- Deployments can ReplicaSets to create multiple instances of the same Pod.
- The Deployment Controller will react to updates and move from actual to desired state at a controlled rate.
- Kubernetes will report status per deployment.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Name of Deployment

Number of Replicas

Container Reference

Container Config (i.e., port, volumes, secrets, env, ...)

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
```

Name of the service

Reference to Pods that should be grouped

How is the service made available?

Other Kubernetes Objects

- ReplicaSet
- Secret
- ConfigMap
- PersistentVolume
- Job
- …

kubectl

- Command line tool to control Kubernetes clusters
- Corresponding cluster is configured in kubeconfig
- Communicates over REST API with cluster controller
- Deploy and manage all parts of the cluster
- Usually work together with the .yml definitions

Conclusion

Interesting Connection Points

- DevOps
- Serverless Computing
- Infrastructure as a Service
- Logging/Monitoring
- Stateless Stream Processing

After today's lecture, you can...

- describe differences in (historic) server hosting styles
- name challenges in service hosting
- explain basic concepts of Docker
- create and run your own Docker image
- deploy a distributed system using Docker Compose
- name high-level concepts and terminology of K8s