07. Container Orchestration

Microservices

Each microservice implements a small part of the system, and runs in a container of it own. The containers communicate using well-known network protocols, which allows for a very high degree of separation of concerns. If one container goes down, it can be restarted without affecting other parts of the system. Scaling up and down is easy, just start up some containers or shut them down.

Problems with microservices

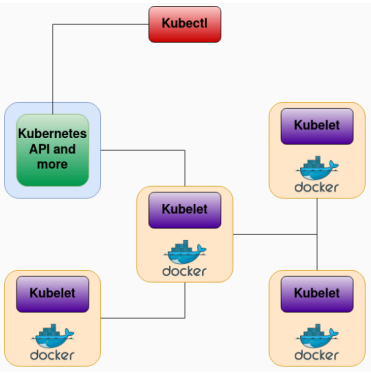
It would take a lot of infrastructure to make this work: setting up networking between containers, start each container running, monitor each container and restart if needed.

Kubernetes

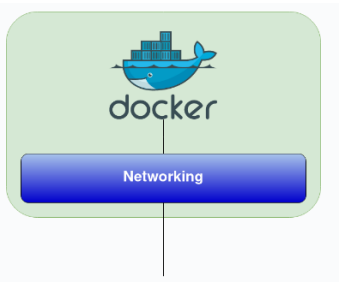
Kubernetes is a container orchestration system, it can automate almost everything previously mentioned for you.

First, we need some physical computers (nodes). Some will be **worker nodes** (which run the containers), other will serve as **control planes** (which manage the containers). The control plane runs the **API**, and each worker runs our containers and a management programme called a **Kubelet**.

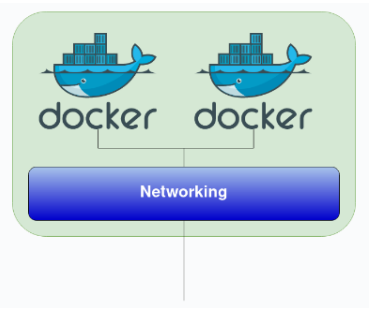
We can control the entire cluster using a **client application** called **Kubectl**, which talks to the PI running on the control plane.



Kubernetes Nodes



Pod



Pod with sidecar containers

The fundamental unit of a Kubernetes cluster is an abstraction of a container called a **Pod.** Pods can contain multiple containers (**sidecar containers**), but generally each pod contains a single container.

Managing a cluster

Kubernetes runs a range of servers on both the control plan and worker nodes to allow pods to be managed and controlled. The things that Kubernetes manages are called **objects**. We specify objects using **YAML documents** which are read by Kubectl.

Creating a pod

```
apiVersion: v1
kind: Pod
metadata:
   name: my-pod
spec:
   containers:
   - name: my-container
         image: some-vendor/some-image
         ports:
         - containerPort: 8080
```

ReplicaSets

Creating a pod directly is very rare, we generally want to create n pods of the same type. This is called a **ReplicaSet**. If one pod starts, it is automatically restarted.

Labels and selectors

ReplicaSets don’t actually maintain a list of all their member pods. Instead, pods store key-value pairs called **labels**. The ReplicaSet finds all of its pods by running queries called **selectors** on these labels.

Creating a ReplicaSet

```
apiVersion: v1
kind: ReplicaSet
metadata:
  name: my-replicaset
  labels:
    app: some-system
    part: frontend
spec:
  replicas: 5
  selector:
    matchLabels:
      part: frontend
```

```
template:
  metadata:
    part: frontend
  spec:
    containers:
    - name: my-container
      image: some-vendor/some-i
mage
      ports:
      - containerPort: 8080
```

Services and ingresses

A service is a network endpoint, pods talk to a service, which redirects to other pods. Services should use selectors to figure out which pods their inputs should be forwarded to. Pods sending a message only need to know the IP address of the service.

A related concept is an Ingress, which provides connectivity to the outside world.

Services

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  replicas: 5
  selector:
    part: frontend
    ports:
    - name: sample-port
      protocol: TCP
```

```
        port: 120
        targetPort: 8080
```

Other important objects

- **Volumes** provide storage shared between pods.
- **ConfigMaps** provide key-value pairs (similar to a YAML document) used to store cluster-wide config information.
- **PersistantVolumes** and **PersistantVolumeClaims** are used to allow pods to store information permanently.
- **Secrets** are similar to ConfigMaps, but can be used to store secret information (e.g. encryption keys, login credentials, ...).

Creating a simple volumes

```
apiVersion: v1
kind: Pod
metadata:
   name: my-pod
spec:
   containers:
   - name: my-container
        image: some-vendor/some-image
        ports:
        - containerPort: 8080
   volumes:
   - name: example-volume
      emptyDir:
        sizeLimit: 1Gi
```

Deployments

An abstraction over ReplicaSets, this is generally what we work in practice. Deployments create and manage a ReplicaSet automatically. They allow for rolling updates, where an old container is slowly replaced with a new one by scaling down an old replica set and scaling up a new one.

```
apiVersion: v1
kind: Deployment
metadata:
    name: my-deployment
    labels:
        part: frontend
spec:
    replicas: 2
    selector:
        matchLabels:
            part: frontend
```

```
template:
    metadata:
        part: frontend
    spec:
        containers:
        - name: my-container
                image: vendor/image
                ports:
                - containerPort: 8080
```