Zurich University of Applied Sciences

School of Engineering

InIT Institute of Applied Information Technology

ORCH – Container Orchestration

Prof. Dr. Thomas M. Bohnert

Dr. Giovanni Toffetti

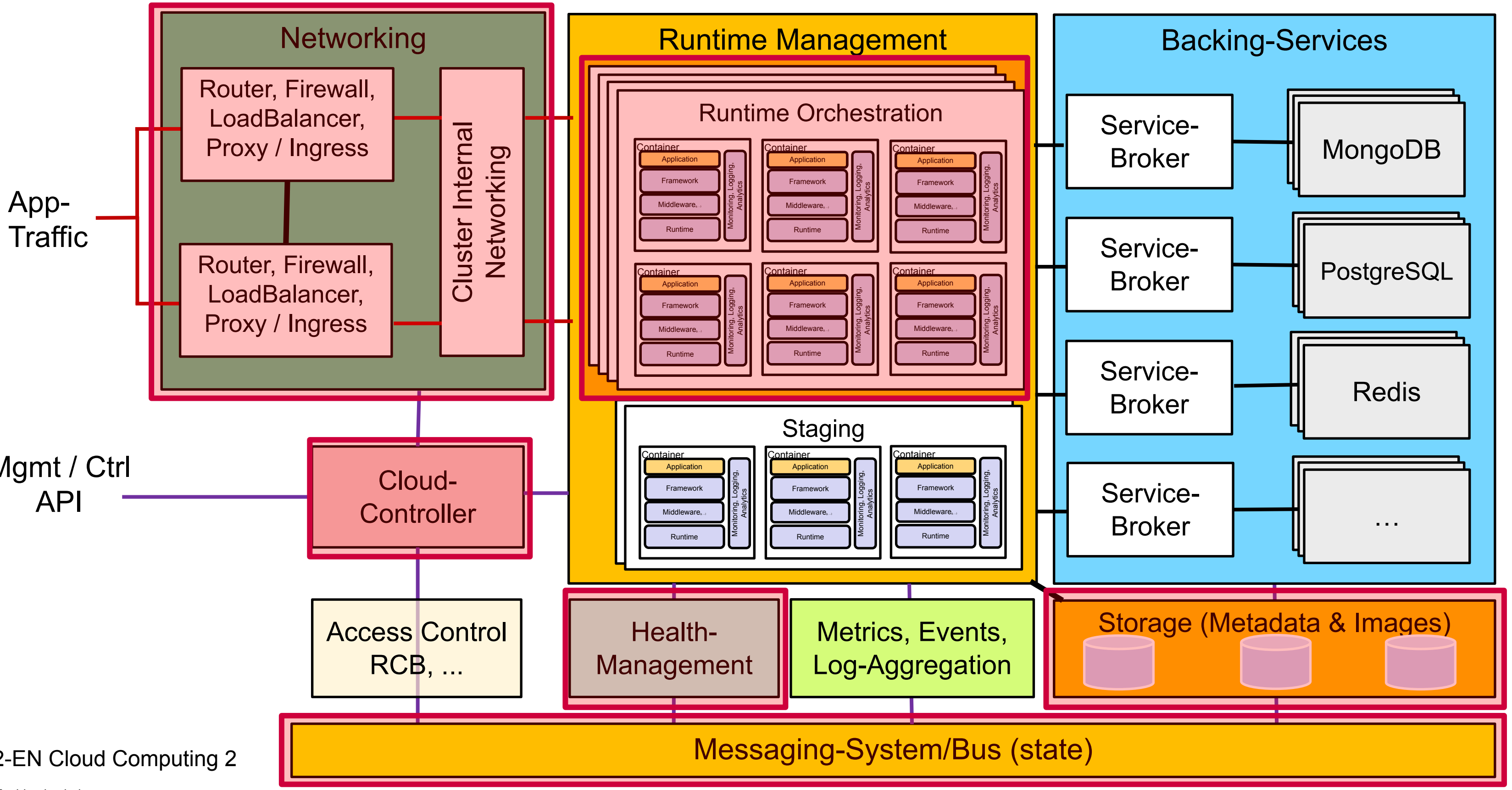
Christof Marti

CCP2-EN Cloud Computing 2

Content

- Container Orchestration
- Kubernetes Core Concepts
- Using Kubernetes

Recap: PaaS Core Architecture Components



MVP for a Container Orchestration Platform

Mandatory components to create Minimal Viable Product:

- **Container Orchestrator**

  We consider the runtimes to be containers

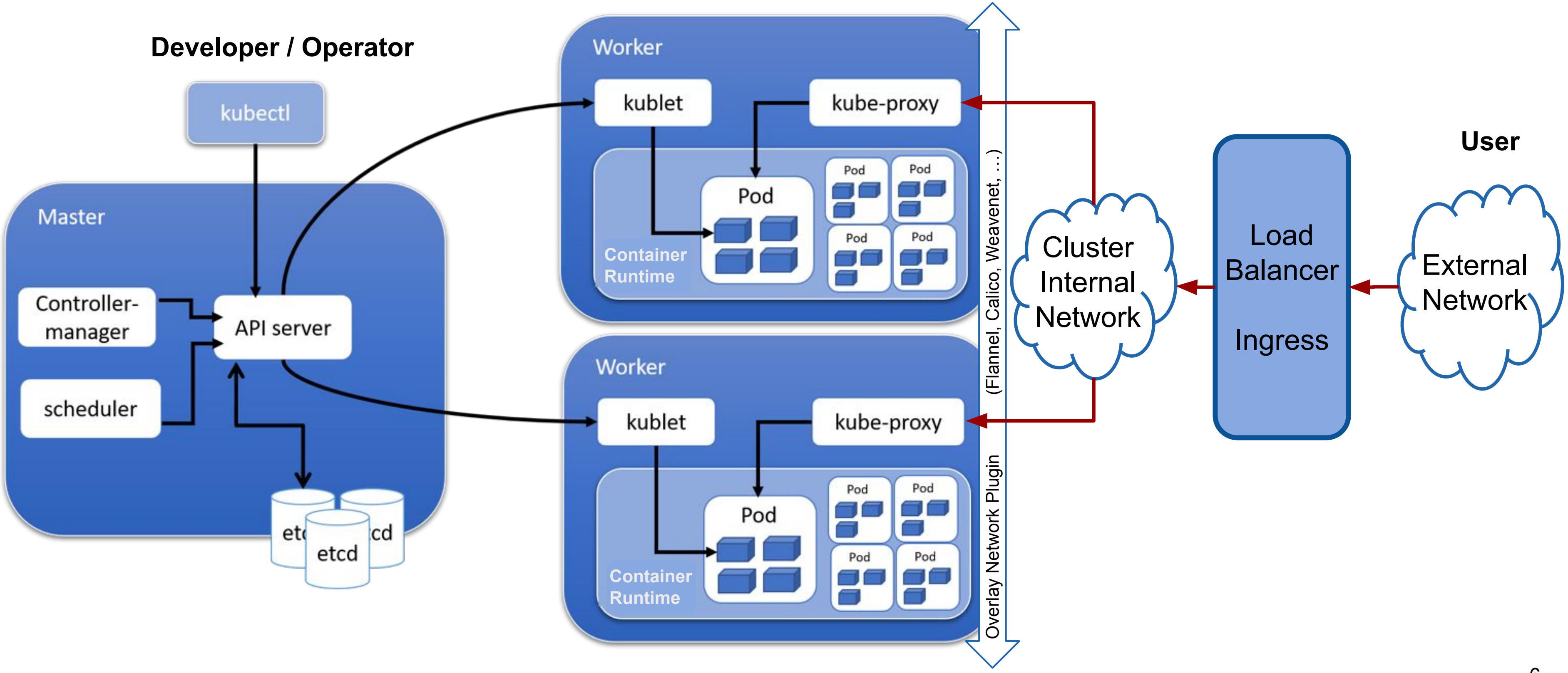
  - **Scheduler** for placing and managing containers on distributed nodes
    - deployment, starting, scaling, stopping, disposal of containers / applications
  - **Shared state** (e.g. distributed DB/KV-Store or Messaging-Bus)
    - Represents the *actual state* of the system vs. the *desired state* in the metadata
  - **Controllers** monitoring container / application health
  - **Mgmt-API** to submit requests get results
- **Storage** local and distributed, ephemeral and persistent
  - for image cache, application runtime and application data
- **Networking**
  - internal connection between applications / services
  - external connection to make it public available

Kubernetes

- Platform for automating deployment, scaling and management of containerized applications.
- Initially written and designed by Google
  - First announced 2014-09
  - Kubernetes v. 1.0 released 2015-07-21
  - At the same time Google and the Linux Foundation create the Cloud Native Computing Foundation (CNCF) and transfer Kubernetes to it
- Borrows heavily from Google's long experience with managing containers, namely Google's Borg System
- Adopted by RedHat for OpenShift, CoreOs for Tectonic, Rancher labs for Rancher, …
- Used by: Google Container Engine, Rackspace, Azure, OpenShift, CloudFoundry …

Kubernetes (κυβερνήτης): Greek for "helmsman" or "pilot", pronounced 'koo-ber-nay'-tace'

Anatomy of a cluster



source: https://gonorthforge.com/kubernetes-and-the-cloud-native-evolution-on-telecom/

Zurich University of Applied Sciences

zh aw School of Engineering

InIT Institute of Applied Information Technology

Anatomy of a cluster

- Components in the Master node(s) (aka Control Plane)
  - **API Server** (kube-apiserver) - Serves the K8s API using JSON/HTTP (external and internal). Scales horizontally by deploying on-demand instances. Possible to run several instances and balance traffic between them.
  - **etcd** - Key/value store keeping the configuration data, representing the overall state of the cluster at any given point of time.
  - **Scheduler** (kube-scheduler) - Selects on which node an unscheduled Pod should run on based on resource availability (pluggable).
  - **Controller Manager** (kube-controller-manager) - Process in which core controllers run, such as *Replication Controller* and *DaemonSet Controller* (Job and Node control).
    - *Cloud Controller Manager* (cloud-controller-manager) - Links a cluster to a Cloud Provider API, separates components that interact with that cloud platform from components that only interact with the kubernetes cluster (Node-VM, Cluster-Routing, Cluster-LoadBalancing)

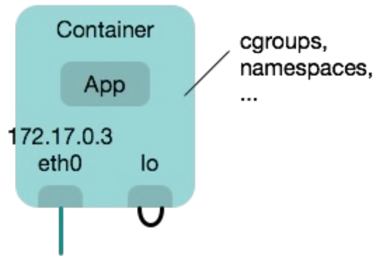
Anatomy of a cluster

- Components in the Worker nodes (aka Kubernetes nodes, Minions)
  - **Kubelet** (kublet) - Running state of each node, i.e. ensuring that all containers on the node are healthy.
  - **Kube-proxy** (kube-proxy) - A network proxy and load balancer. Routing traffic to the appropriate container, based on IP and port number of the incoming request. Allows network communication to Pods from network sessions inside or outside a cluster.
  - **Container Runtime** - Responsible for running containers: Docker, containerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface)
  - **Overlay network** – Connect containers on different nodes on a flat network. Provided by plug-ins (e.g., Flannel, Calico, Weave, …)

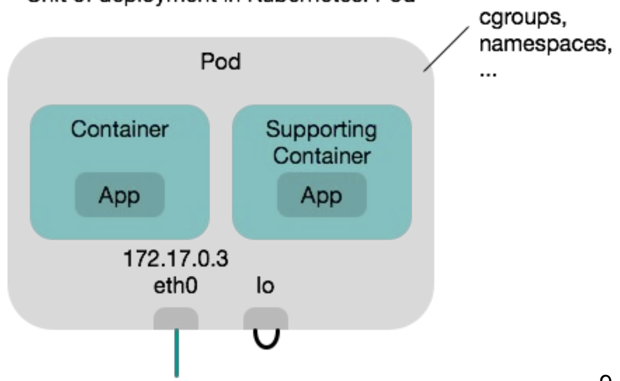
Kubernetes Concepts – Pods

- The atomic unit of deployment in Kubernetes is the Pod.
  - can be scaled/replicated horizontal to multiple nodes
- A Pod contains one or more containers. The common case is a single container.
- If a Pod has multiple containers
  - Kubernetes guarantees scheduling on the same cluster node.
  - The containers share the same environment
    - IPC namespace, shared memory, volumes, network stack, etc.
    - IP address
  - If containers need to talk to each other within the Pod, they can simply use the localhost interface.
  - Containers in the same Pod “live and die” together
- Pods are ephemeral: if a node fails, Pods on the node do not get automatically rescheduled, they need to be restarted

Unit of deployment in Docker: Container

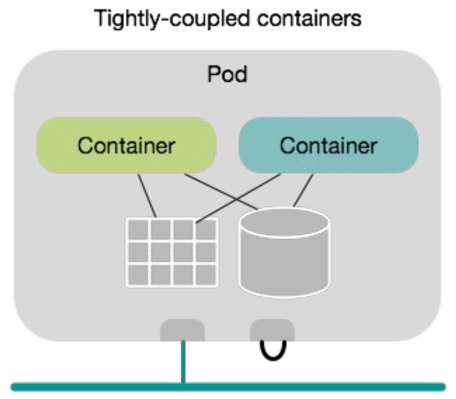


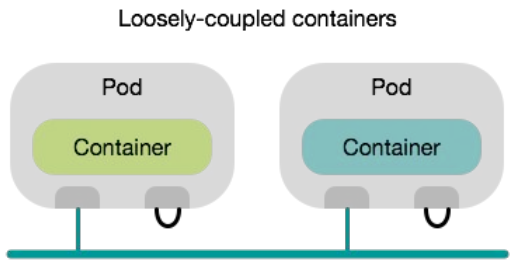
Unit of deployment in Kubernetes: Pod



Kubernetes Concepts – Pods

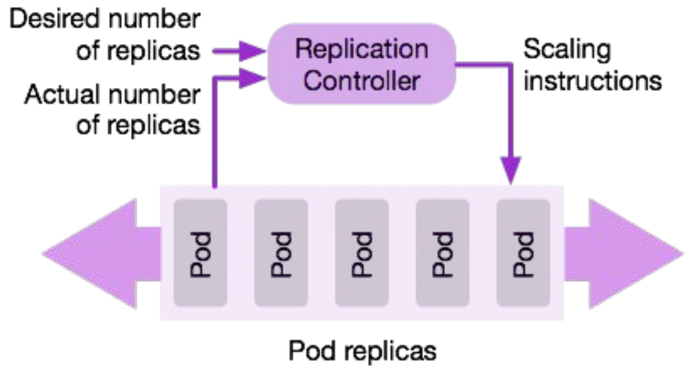
- The Pod abstraction enables the user to choose between loosely coupled or tightly coupled containers.
- Example use-cases for tightly coupled containers:
  - A web container supported by a helper container that ensures the latest content is available to the web server.
  - A container with a log scraper, sending the logs off to a logging service somewhere else.
  - A container with a support container managing secure connection to other containers (→ service mesh)
- These tightly coupled support containers are often called ***sidecar container***

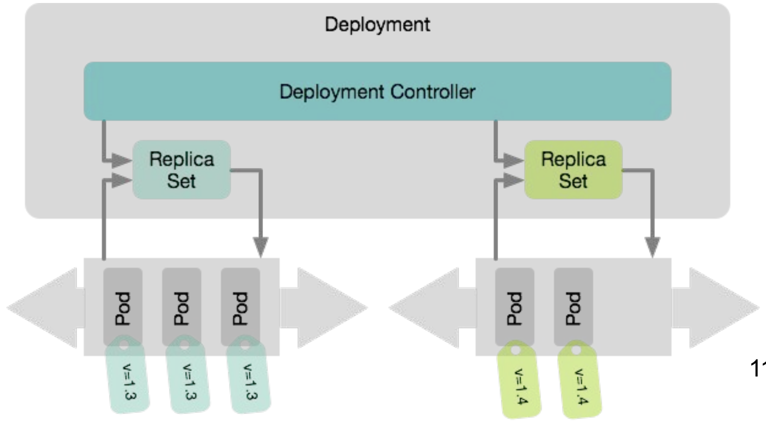




Kubernetes Concepts – Replication

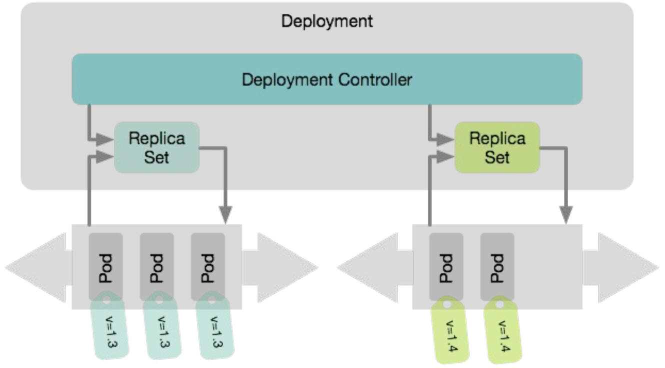
- A **Replication Controller** deploys a desired number of replicas of a pod definition
  → horizontal scaling, enhance resilience
- It ensures that the correct number of replicas is always running, stops unresponsive and starts new pods (desired vs. actual state)
- Replication Controllers are replaced by the more flexible **Replica Sets** which allow specifying the pods to replicate using **Labels**.
- Deployments (see next) may use several active Replica Sets to perform rolling updates or rollbacks





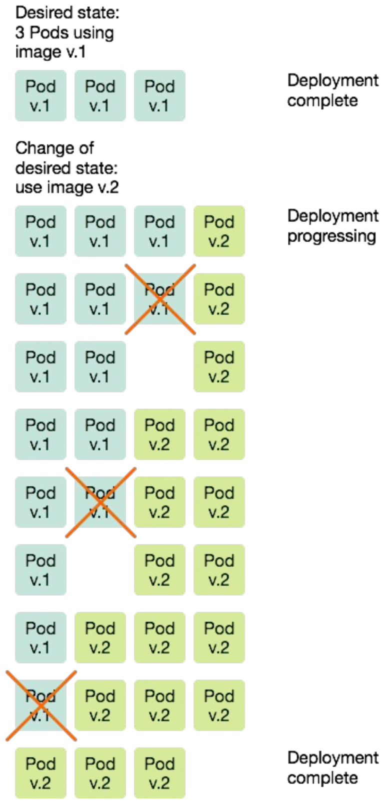
Kubernetes Concepts – (Application) Deployment

- The concept of an Application **Deployment** combines *Pods* and *Replication* and extends it with a *Software Release Management* process.
  - Create an initial Release
  - Updating to a new Release
  - Rolling back to a previous Release
  - Deleting a Release
- Supports different Strategies to update and rollback
  - *Recreate* → kill existing pods and bring up new ones faster, but application has downtime
  - *Rolling Update* → gradually brings up new and kills old pods slower, but application stays available
- A **Deployment Controller** manages the process
- Uses multiple **Replica Sets** to implement update/rollback



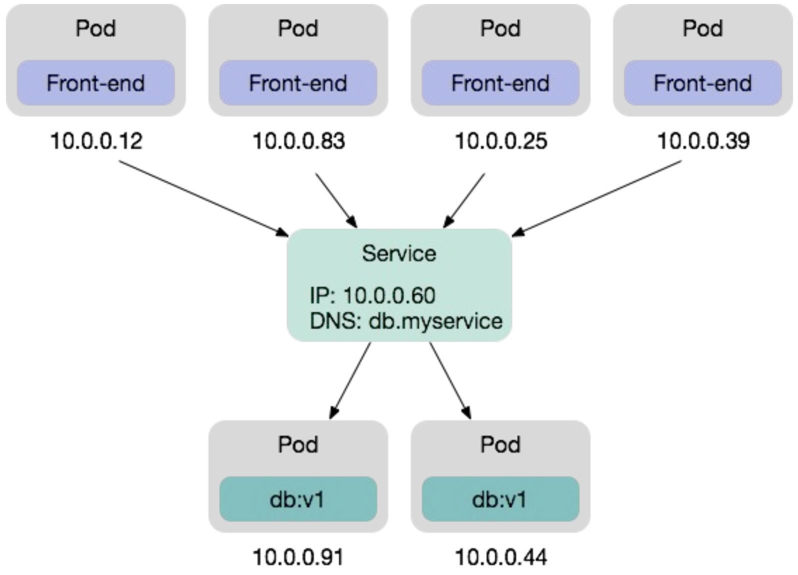
Kubernetes Concepts – Deployment

- In a **rolling update,** the service is never interrupted.
- To trigger a rolling update, the user simply updates the desired state of the deployment.
  - For example, specify a different version of the Pod image.
  - The Deployment creates a new Replica Set.
  - The Deployment Controller creates a new Pod in the new Replica Set, and when successful terminates a Pod in the old Replica Set.
  - This is repeated until no old Pods are left.


Desired state: 3 Pods using image v.1
Pod v.1
Deployment complete
Change of desired state: use image v.2
Pod v.2
Deployment progressing
Deployment complete

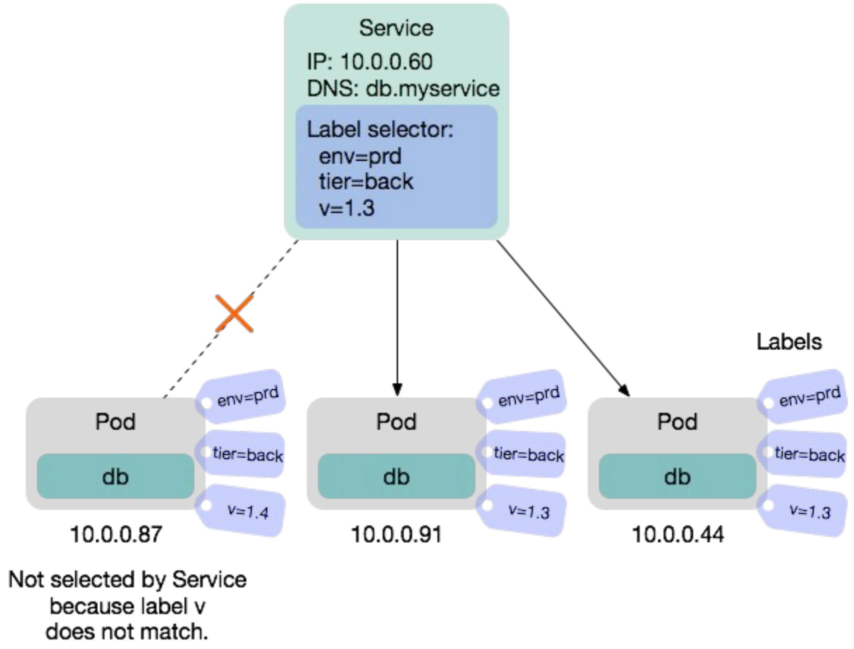
Kubernetes Concepts – Service

- When replacing, scaling or upgrading Pods, they receive new IP addresses every time.
  - Suppose a two-tier app with a front-end Pods talking to back-end Pods. The front-end Pods cannot rely on the IP addresses of the back-end Pods.
- A **Service** provides a reliable networking endpoint for a set of Pods.
  - The front-end talks to the *reliable IP* of the service.
  - The IP address is mapped to a more expressive *service name* using a DNS service
  - The Service *load-balances* all requests over the back-end Pods behind it.
  - The Service keeps track of which Pods are behind it.



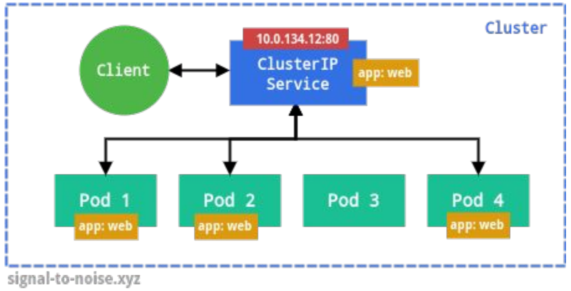
Kubernetes Concepts – Label & Label Selector

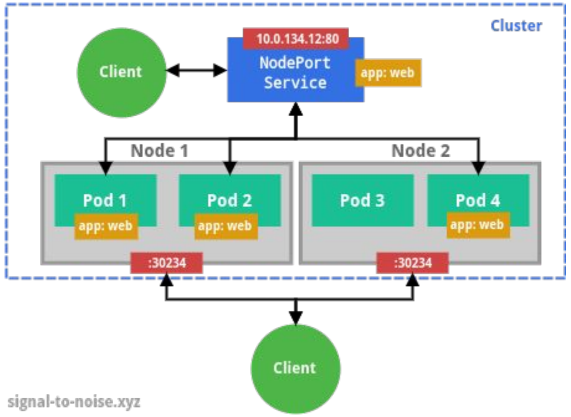
- To specify which Pods belong to a service, one uses **Labels**.
  - Labels are key-value pairs attached to a Kubernetes object.
  - Key and value can be freely chosen.
  - An object may have several labels. The same label may be attached to multiple objects.
- When defining the Service, one specifies a **Label Selector** which is a set of conditions on the label key-values.
  - Pods matched by the Label Selector are connected to the Service.
- Beside Service, labels will be used by many K8s objects (Replica Set, Persistent Volume, Ingress, ...)



Exposing a Service – ServiceType

- As explained, a Service provides a *reliable network endpoint* for a set of pods (i.e. Deployment)
- To define how this network endpoint is selected a **ServiceType** has to be specified:
  - **ClusterIP** (default)
    Exposes the Service on an internal IP in the cluster.
    This type makes the Service only reachable
    from within the cluster.
  - **NodePort**
    Exposes the Service on the same port (NodePort)
    of each selected Node in the cluster using NAT.
    Makes a Service accessible from outside the cluster using a
    Superset of ClusterIP.

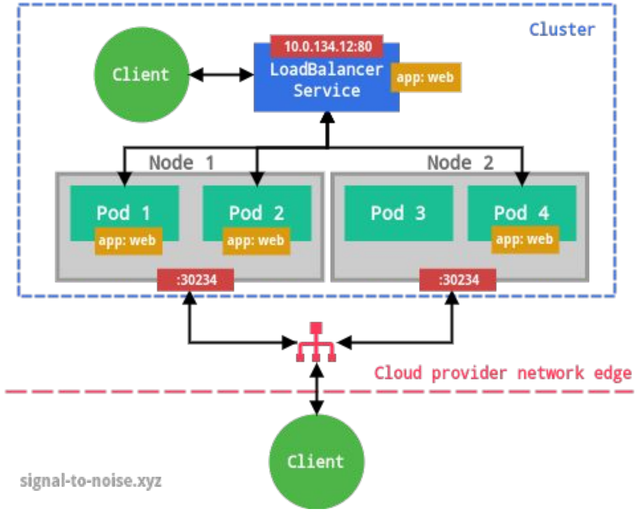




Exposing a Service – ServiceType

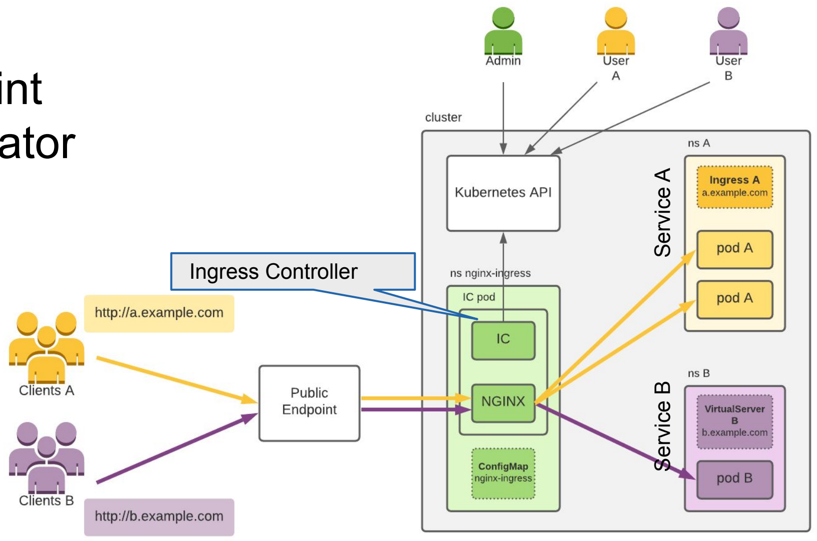
- **LoadBalancer**
  Exposes the service externally using a cloud provider load balancing service (if supported) and assigns a fixed external IP to the service.
  ClusterIP and NodePort to which the traffic is routed are created automatically.

- **ExternalName**
  Maps the service to a DNS CNAME record (specified by externalName in the spec).
  No proxy is used.
  It is essentially a redirect to the CNAME.
  This type requires v1.7 or higher of kube-dns



Exposing a Service – Ingress

- An **Ingress** is an additional concept to expose a Service externally.
- It forwards (proxies) HTTP(S) traffic based on domain/path to specific service endpoint.
- An **Ingress Controller** reads the desired state (ingress objects) from the Kubernetes API and configures the proxy table.
  https://domain/path → Service Endpoint
- Additionally, it may also work as an SSL endpoint
- The Ingress is managed by a Cluster Administrator



Kubernetes Concepts – Volumes

- A **Volume** is storage shared between containers in the same Pod
- Lifetime: same as Pod (outlives containers if restarted) unless using persistent volume solutions from underlying infrastructure
- Supported types:
  - **emptyDir**: self-explanatory, erased at Pod deletion
  - **hostPath**: path from host machine, persisted with machine lifetime
  - **gcePersistentDisk, awsElasticBlockStore, azureFileVolume**: cloud vendor volumes, independent lifecycle
  - **secret**: used to pass sensitive information, such as passwords, to Pods. You can store secrets in the Kubernetes API and mount them as files for use by Pods without coupling to Kubernetes directly. secret volumes are backed by tmpfs (a RAM-backed filesystem) so they are never written to non-volatile storage
  - **nfs** : network file system, persisted
  - Many more: iscsi, flocker, glusterfs, rbd, gitRepo ...

Kubernetes Concepts – Volumes

- **Persistent Volume (PV)**
  Volumes shared by the cluster of a specific type
  - **static**: volumes explicitly created by a Cluster Admin
  - **dynamic**: using a Storage Class configured by the admin
    will create a specific volume on demand (ephemeral or persistant)
- **Persistent Volume Claim (PVC)**
  - Request to bind a Pod / Deployment to a PV by the user
  - Specifying Volume (static) or Storage Class (dynamic) and required size
  - Using default Storage Class if none found or not specified

Using Kubernetes – Principles

- Kubernetes works according to the principle of *Desired State Configuration*
  - The user describes the desired state in a Kubernetes object and applies it to the cluster.
  - The matching Controller compares the desired state to the actual state.
  - And changes the actual state towards the desired state at a controlled rate.
- Kubernetes is idempotent
  - If the same object is applied multiple times, the same result is obtained
  - Nothing is changed if the actual state is already the desired state
  - The Actual state is returned in the status section of the response
- Requests are sent using a REST-API, which is used by many clients
  - kubectl, k9s, oc, …
  - Kubernetes Web UI / Dashbord, Lens, OpenShift Web UI, …
  - SDK for Go, Python, Java, Node, …

Using Kubernetes – Object Resource API

- Kubernetes adopts a consistent **object resource API**
- Every Kubernetes object has some basic fields in its description:
  - **kind** specifies the type of object (pod, deployment, service, ingress, …)
  - **apiVersion** specify the version of the object API used (e.g. apps/v1)
  - The Object **metadata** is similar for all object types
    - it contains information such as the object's **name**, **UID** (unique identifier), an object **version number** (for optimistic concurrency control), and **labels** (key-value pairs).
  - **spec** is used to describe the desired state of the specific object
  - **status** provides read-only information about the current (actual) state of the object

- The object resources (manifests) are usually written in YAML (or JSON)
  - Tedious to edit (indentation)
  - But good support by Editors/IDE e.g. VS Code

Using Kubernetes - Example Deployment Manifest

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-nginx
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Deployment name

Replica Set with 3 replicas

Pod may contain multiple container

Container Image id or URL

Exposed port of container

Some helpful resources to read, edit manifests

- Get api-versions: `kubectl api-versions`
- Get list of api-resources (kind): `kubectl api-resources`
- Get info about resource:
  ```
  kubectl explain --api-version=apps/v1 Deployment
  ```
- Get explaination of resource fields:
  ```
  kubectl explain --api-version=apps/v1 Deployment.metadata
  kubectl explain --api-version=apps/v1 Deployment.spec.template
  ```
- Get a hierachical view:
  ```
  kubectl explain --api-version=apps/v1 Deployment --recursive
  ```

Using Kubernetes - Example Service

```
apiVersion: v1
kind: Service
metadata:
  name: my-nginx
spec:
  ports:
    - port: 8080 # port of the service
      protocol: TCP
      targetPort: 80    # port on pod
        nodePort: 30007 # port on node
  selector:
    app: nginx  # reference to deployment
  type: NodePort
```

Sending requests

- Create or update a resource:
  `kubectl` **`apply`** `-f my-nginx-deployment.yaml`

- Display information about a deployment resource:
  `kubectl` **`get`** `deployments my-nginx`
  `kubectl` **`describe`** `deployments my-nginx`
  about a ReplicaSet:
  `kubectl get replicasets`
  `kubectl describe replicasets`

- Deleting a resource:
  `kubectl delete deployment my-nginx`

Using Kubernetes - Persistant Volumes

K8S supports mounting (existing or dynamically provisioned) persistent volumes to Pods in two ways:

- Directly
- Through PersistentVolumeClaim (PVC)

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: test-ebs
spec:
  containers:
  - image: icclab/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /test-ebs
      name: test-volume
  volumes:
  - name: test-volume
    # This AWS EBS volume must already exist.
    awsElasticBlockStore:
      volumeID: <volume-id>
      fsType: ext4
```

```yaml
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  # The cluster must have a
configured storage class
  storageClassName: default
```

Appendix