Kubernetes - Überblick

1. MicroServices, Docker, DockerHosts, Consul
2. Kubernetes
    - k8s Architektur
    - k8s API Objekte
    - k8s Networking, Istio
    - k8s RBAC
3. Kubernetes Deployment / helm / ArgoCD
4. Lens Kubernetes IDE
5. IT/OPS vs DevOPS vs Dev

MicroServices

Software die zur Ausführung einer bestimmten Business-Funktionalität dient.

Ein MicroService ist so verpackt, dass er bereits alles zur Ausführung dieser Funktion enthält und lediglich eine Laufzeitumgebung braucht, in dem er gestartet werden kann.

- SpringBoot-MicroService → Java Runtime,
- Go-MicroService → Go Runtime,...

Docker

Docker ist eine Software, welche die Container-Virtualisierung von Anwendungen ermöglicht. Anwendungen können inklusive ihrer Abhängigkeiten in ein Image gepackt werden. Mittels einer speziellen Engine kann die so verpackte Anwendung dann in einem Docker Container ausgeführt werden.

Container sind selfcontained, dadurch wird eine sehr gute Trennung zwischen Entwicklung und IT/OPS erreicht. Entwickler kümmern sich um die Anwendung, IT/OPS um die Infrastruktur

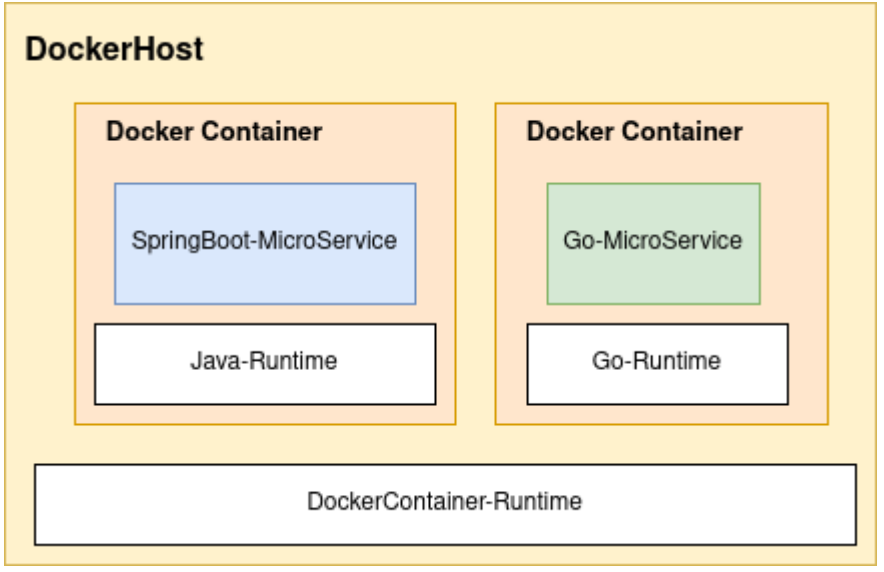
DockerContainer brauchen zur Ausführung eine DockerContainer-Runtime

DockerHosts

DockerHost (cattle-hosts) sind „Wirts“-Rechner mit einer **Docker-Container-Runtime**, auf denen Docker Container ausgeführt werden können.

Wie kommen die DockerImages auf den DockerHosts?

Im build-Step der Gitlab Pipeline wird das DockerImage, das den MicroService enthält, erzeugt und in der DockerRegistry gespeichert. Im deploy-Schritt wird dann **docker-compose** auf dem (Remote)DockerHost aufgerufen und das DockerImage aus dem build-Step installiert.



DockerHosts & Consul

Auf jedem DockerHost läuft zusätzlich ein (Consul)Registrator Service.

Wird ein neuer Docker-Container gestartet, bekommt der Registrator das mit und kann den exposed Port des Containers und den Service-Namen auslesen.

Mit diesen Informationen wird der Service aus dem Docker-Container am Consul registriert.

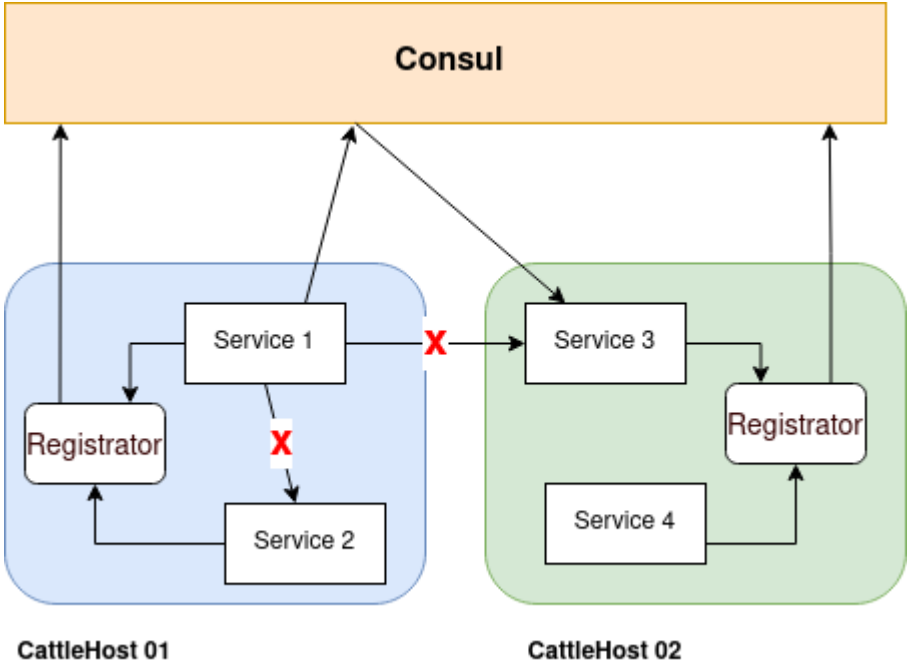
MicroServices sind ja miteinander gekoppelt und müssen miteinander kommunizieren. Ein MicroService weiß jetzt aber nicht, auf welchem DockerHost der andere MicroService installiert ist. Er kennt lediglich den Consul-Namen dieses MicroService.

Consul

Consul ist ein Discovery-Service.

Consul verwaltet Services und Knoten in der Art eines “Verzeichnisdienstes”, also in der Form was läuft wo?

Die Services benutzen die Service-Consul-URLs um miteinander zu kommunizieren.



DockerHosts & Consul Nachteile?

Die Kombination aus DockerHost und Consul ist eine wenig flexible und nicht sehr umfangreiche Lösung.

Consul:

- ist (nur) ein DiscoveryService,
- führt HealthChecks gg. die Service aus,
- könnte Loadbalancing,
- hat kein Interface zum Starten/Stoppen der Service oder von Jobs,
- kann nicht für Ausfallsicherheit sorgen.

Skalierung: Keine oder nur Pseudo-Skalierung möglich

- ist kein Problem des Consul, sondern das es unsere deploy-Pipeline nicht hergibt, eine zweite Instanz eines Service auf dem (einem anderen) DockerHost zu deployen,
- das liegt aber eigentlich auch nicht in der Verantwortung einer gitlab deploy Pipeline.

Kein **Resourcen-Management** auf den DockerHosts.

Exkurs Skalierung

Vertikale Skalierung: mehr RAM, mehr CPU für die Anwendung auf einem Host

Horizontale Skalierung: mehrere Instanzen einer Anwendung auf einem oder mehreren Hosts (erfordert Loadbalancing)

Kubernetes

Griechisch für: Steuermann z. B. eines (Container-)Schiffes

engl.: “**helm**sman”

Auch **k8s** k{ubernete}s (Numeronym)

Was ist Kubernetes?

Eine ursprünglich von Google entwickelte Open-Source-Plattform zur Orchestrierung von Containern. (Nicht ausschließlich Docker-Container!)

Warum Container-Orchestrierung?

- Anwendungen konkurrieren um Ressourcen.
- Anwendungen müssen untereinander kommunizieren können (oder sollen es auch *nicht* dürfen können).
- Anwendungen brauchen ein Lifecycle Management.
- Anwendungen müssen von zentraler Stelle administriert werden können.
- Anwendungen brauchen ein Monitoring.
- Anwendungen muss man skalieren können.
- Anwendungen brauchen ein Loadbalancing.

Kubernetes – 3 Designprinzipien

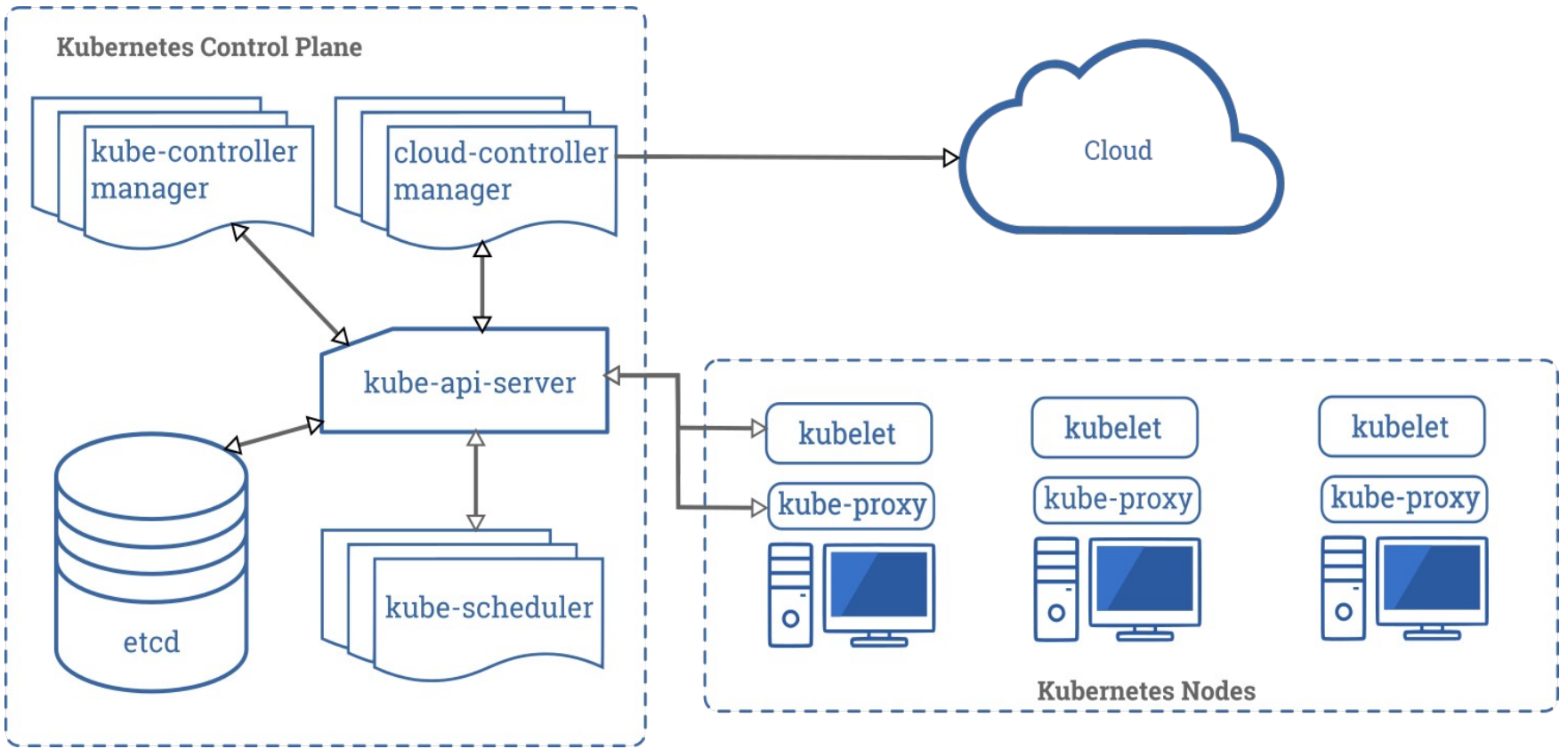
Sicher: k8s sollte den neuesten Best Practices für die Sicherheit entsprechen. (Netzwerksicherheit, Ausfallsicherheit, Zugriffsrechte)

Anwenderfreundlich: k8s sollte mit wenigen einfachen Befehlen zu verwenden sein.

Erweiterbar: k8s sollte keinen Anbieter bevorzugen und über eine Konfigurationsdatei angepasst werden können.

- k8s ist weder an einen bestimmten Cloud-Provider, ein bestimmtes Betriebssystem oder eine bestimmte Container-Technologie gebunden.

Kubernetes Architektur - Übersicht

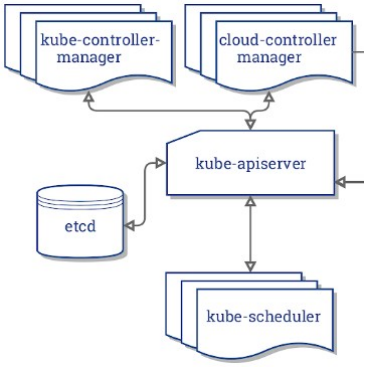


Kubernetes Control-Plane

Hier befinden sich die Kubernetes-Komponenten zur **Steuerung des Clusters** sowie Daten zum **Status und zur Konfiguration** des Clusters. Die Kernkomponenten von Kubernetes übernehmen die wichtige Aufgabe, sicherzustellen, dass die Container in ausreichender Anzahl und mit den erforderlichen Ressourcen ausgeführt werden.

Die Control Plane steht in ständigem Kontakt mit den Rechenmaschinen und stellt sicher, dass der Cluster auf die konfigurierte Weise ausgeführt wird.

Control-Plane Komponenten



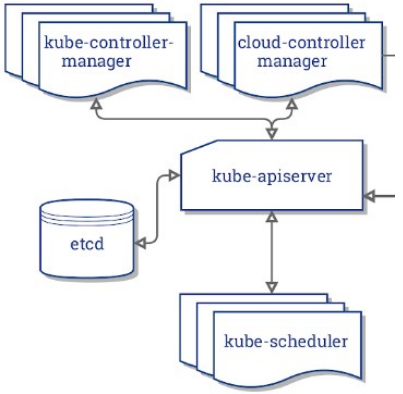
kube-api-server:

- Interaktion mit dem Kubernetes-Cluster durch die Kubernetes-API. Die Kubernetes-API ist das **Frontend der Kubernetes Control Plane** und verarbeitet interne und externe Anfragen. Der API-Server ermittelt, ob eine Anfrage gültig ist, und verarbeitet sie dann. Die API kann über REST-Aufrufe, über die Befehlszeile kubectl oder über andere Befehlszeilen-Tools wie kubeadm aufgerufen werden.

kube-scheduler:

- In welchem Zustand befindet sich der Cluster? Wenn neue Container benötigt werden, wo passen sie hin? Dies sind die Fragen, die im Kubernetes-Scheduler beantwortet werden.
- Der Scheduler berücksichtigt den Ressourcenbedarf eines Pods, wie die CPU oder den Arbeitsspeicher, sowie den Zustand des Clusters. Anschließend wird der Pod für einen geeigneten Rechenknoten geplant.

Control-Plane Komponenten



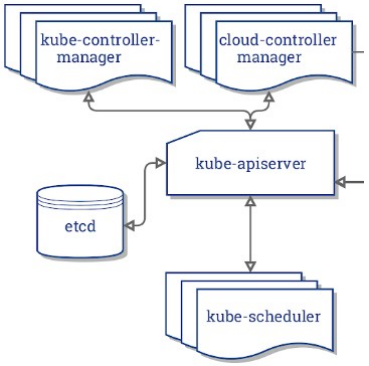
kube-controller-manager:

- Controller sorgen dafür, dass der Cluster tatsächlich ausgeführt wird. Der kube-controller-manager enthält mehrere ineinander verschachtelte Controller-Funktionen. Ein Controller befragt den Scheduler und stellt sicher, dass die richtige Anzahl von Pods ausgeführt wird (NodeController). Wenn ein Pod ausfällt, springt ein anderer Controller ein (ReplicationController). Ein Controller verbindet die Services mit den Pods, sodass alle Anfragen an die richtigen Endpunkte gelangen. (EndpointsController) Außerdem gibt es Controller, die Accounts und API-Zugriffs-Token erstellen.

etcd

- Speicherdatenbank für Schlüsselwerte, dort befinden sich die Konfigurationsdaten und Informationen zum Status des Clusters. Etcd bildet die „Ultimate Source of Truth" des Clusters.

Control-Plane Komponenten

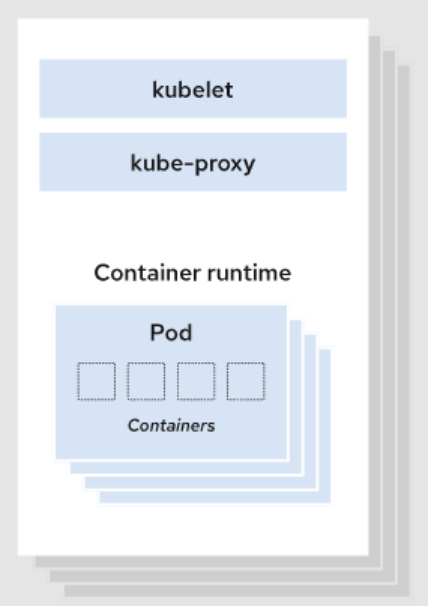


Cloud-controller-manager:

führt Controller aus, die mit den entsprechenden Cloud-Anbietern interagieren.

Worker-Nodes

- Kubernetes verfügt über mindestens einen Worker-Node auf dem die Anwendungen ausgeführt werden. Aber ein Cluster hat natürlich besser mehrere Worker-Nodes
- Worker-Nodes sind reale oder virtuelle Rechner im Rechenzentrum oder in der Cloud
- Auf jedem Worker-Node ist eine Container-Runtime installiert, damit die Container dort laufen können
- Auf jedem WorkerNode gibt es *n* Pods in dem die Container deployt werden



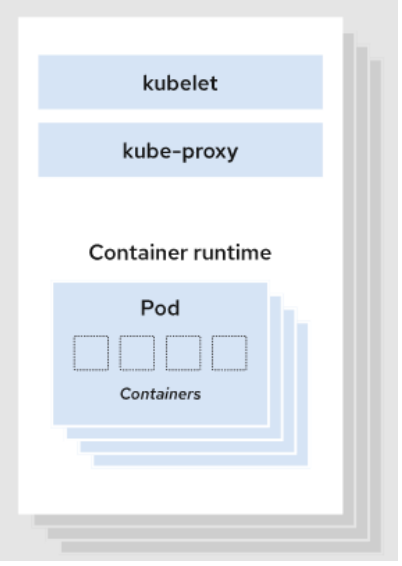
Worker-Node Komponenten

kubelet:

- Jeder Rechenknoten enthält ein kubelet, eine Anwendung, die mit der Control Plane kommuniziert. Das kubelet stellt sicher, dass die Container in einem Pod ausgeführt werden. Wenn die Control Plane einen Vorgang in einem Knoten benötigt, führt das Kubelet die Aktion aus. kubelet ist somit der control-plane Agent.

kube-proxy:

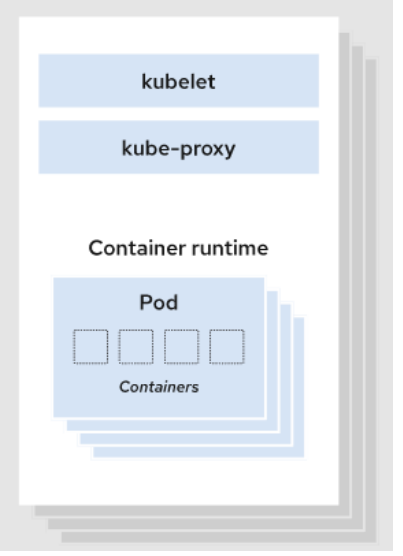
- Jeder Rechenknoten enthält auch den kube-proxy, einen Netzwerk-Proxy für die Kubernetes-Netzwerk-Services. Der kube-proxy verwaltet die Netzwerkkommunikation innerhalb oder außerhalb Ihres Clusters.



Worker-Node Komponenten

Container Runtime Engine:

- Zur Ausführung der Container verfügt jeder Rechenknoten über eine Container Runtime Engine. Docker ist nur ein Beispiel, aber Kubernetes unterstützt auch andere Runtimes, die mit der Open Container Initiative kompatibel sind, z. B. rkt und CRI-O.
- Pods sind die „Orte“ in denen die Container ausgeführt werden.



Kubernetes API-Objekte

Kubernetes-Objekte sind persistente Entitäten im Kubernetes-System. Kubernetes verwendet diese Entitäten, um den Zustand des Clusters darzustellen.

Ein Kubernetes-Objekt ist ein „record of intent“ – sobald das Objekt erstellt ist, arbeitet das Kubernetes ständig daran, sicherzustellen, dass das Objekt vorhanden ist.

Zum Erstellen, Ändern oder Löschen von Kubernetes-Objekten muss die Kubernetes-API verwendet werden.

Deklaratives Management

Die Konfiguration der Kubernetes-API Objekt wird in .yaml Dateien gespeichert.

Die grundlegende Struktur ist:

apiVersion: <die API-Group und die Version der verwendeten API>

kind: <Typ des API-Objekts>

metadata: <Metadaten wie Name des API-Objekts und der Namespace>

spec: <Spezifikation des API-Objekts, wie das Container Image>

API-Objekte: Namespaces

Namespaces sind die Ordnungseinheiten innerhalb eines Clusters, sozusagen virtuelle Cluster, die von demselben physischen Cluster unterstützt werden. Diese virtuellen Cluster werden als Namespaces bezeichnet.

Es gibt Kubernetes eigene (admin)Namespaces.

Konvention: Jeder BoundedContext hat seinen eigenen Namespace im Kubernetes Cluster.

API-Objekte: Pod

Ein Pod ist die kleinste und einfachste Einheit im Kubernetes-Objektmodell. In diese Einheit wird deployt und er steht für eine einzelne Instanz einer Anwendung. Jeder Pod besteht i.d.R. aus einem Container (oder eng miteinander verbundenen Container). Pods können mit persistentem Storage verbunden werden, um zustandsbehaftete Anwendungen auszuführen.

- hat jeweils eine eigene, **kubernetes-interne IP**
- diese Einheit kann skaliert werden
- kann weitere sogenannte Init- oder **Sidecar**-Container beinhalten

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

API-Objekte: Deployment

Das Deployment ist eine Workload-Resource.

Deployment ist ein HighLevel Object, das sowohl die Definition des Pods und eines ReplicaSets enthalten kann.

Es beschreibt einen gewünschten Zustand, und der Deployment-Controller ändert den Ist-Zustand und kontrolliert auf den gewünschten Zustand.

Im Pod ist enthalten, welches Container-Image installiert werden soll.

Im ReplicaSet wird bestimmt wie viele Kopien dieses Pods gleichzeitig laufen sollen.

Dieses Objekt benutzen wir um unsere stateless (SpringBoot)Services zu deployen.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
  labels:
    app: nginx
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: myserver
        image: nginx:latest
        ports:
        - containerPort: 8080
```

API-Objekte: CronJob

Ein CronJob ist auch eine Workload-Resource und kreiert sich wiederholende Jobs.

Ein Job startet ein oder mehrere Pods und stellt sicher, dass eine definierte Anzahl dieser erfolgreich beendet werden.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - "date; echo Hello from pod"
          restartPolicy: OnFailure
```

API-Objekte: ConfigMap

Eine ConfigMap ist zum Speichern nicht vertraulicher Daten in key-value pairs.

Pods können ConfigMaps als environment variables, command-line arguments oder als Konfigurations-Dateien in einem Volume konsumieren.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: index
data:
  index.html: |
    <h1>Hello</h1>
  bye.html: |
    <h1>Good Bye!</h1>
---
apiVersion: v1
kind: Pod
metadata:
  name: www
spec:
  containers:
  - name: www
    image: nginx:latest
    ports:
    - containerPort: 8080
    volumeMounts:
    - name: website
      mountPath: /usr/share/nginx/html
    imagePullPolicy: IfNotPresent
  volumes:
  - name: website
    configMap:
      name: index
```

API-Objekte: Secrets

Mit Secrets werden die vertraulichen Informationen wie Passwörter, OAuth-Token und ssh-Schlüssel verwaltet.

Secrets unabhängig vom Pod zu speichern ist sicherer und flexibler als das explizite Speichern innerhalb einer Pod-Definition oder im Container-Image, weil sie dadurch nicht beim Anzeigen eines Pods exposed werden können.

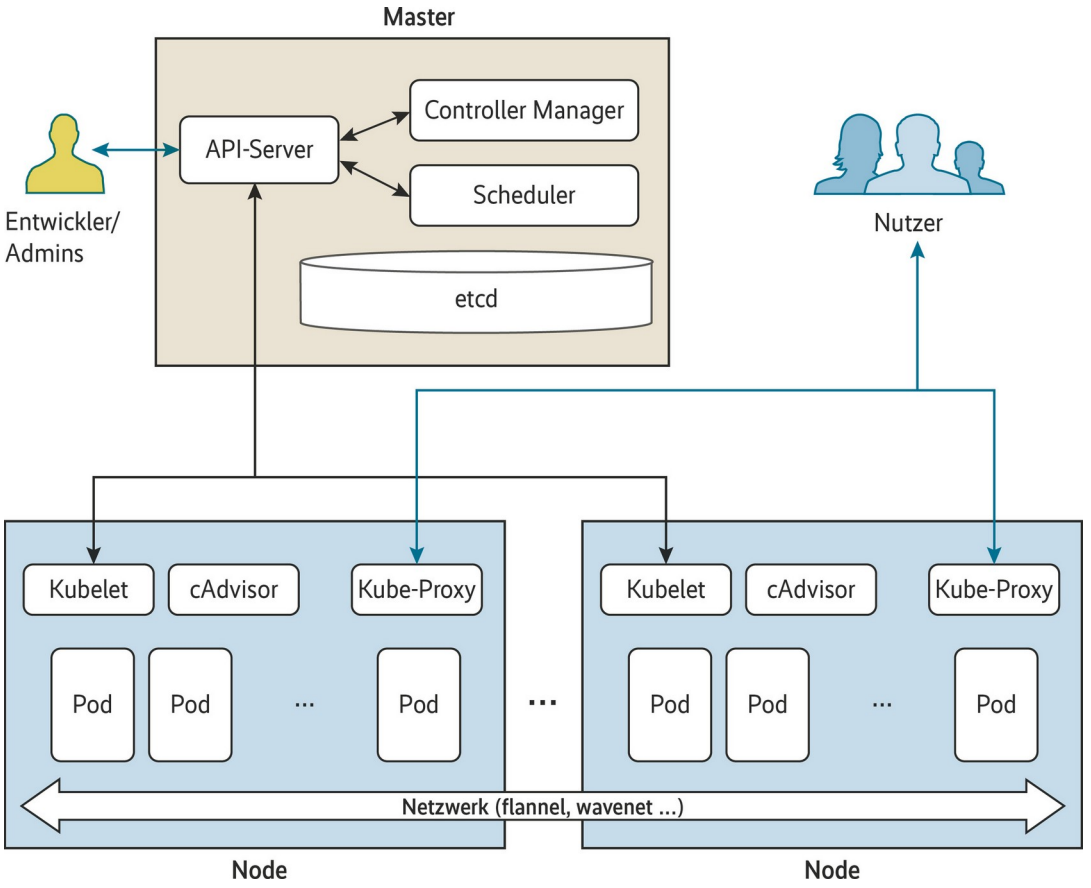
Secrets werden base64 encoded angegeben. Per default speichert Kubernetes die Secrets nur encoded aber nicht encrypted im **etcd**. Der Zugriff darauf kann aber mittels **RBAC** geschützt sein.

Kubernetes ist offen für andere Credential-Provider Lösungen

In der xxx sind die sensitiven Daten per **sops** in den secrets.yaml Dateien verschlüsselt, damit diese Daten zumindest so verschlüsselt im Git Repository sind. Während des Deploy Vorgangs werden diese Daten denn entschlüsselt und dann wieder lediglich nur base64 encoded an Kubernetes übergeben.

```
apiVersion: v1
kind: Secret
metadata:
  name: my-api-token
  namespace: default
  labels:
    app: api-consumer
data:
  apiToken: bXlBcGlUb2tlblN1cGVyR2VoZWltCg==
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-env-pod
spec:
  containers:
  - name: container-w-env-secret
    image: alpine:latest
    imagePullPolicy: IfNotPresent
    command:
    - sh
    - -c
    - "while true; do date >> /srv/log ; sleep 1 ; done "
    env:
      - name: SECRET_APITOKEN
        valueFrom:
          secretKeyRef:
            name: my-api-token
            key: apiToken
```

Kubernetes Networking



Networking ist der zentrale Part von Kubernetes!

Es gibt dabei 4 verschiedene Aspekte:

- Container-zu-Container-Kommunikation
- Pod-zu-Pod-Kommunikation
- Pod-zu-Service-Kommunikation
- External-zu-Service-Kommunikation

Networking - API-Objekt: Service

Ein Service ist eine Netzwerk-Resource.

Ist eine Abstraktion um eine in einem oder mehreren Pods laufende Applikation als Netzwerkdienst bereitzustellen (exposen).

Der DNS-Name ist der Servicename, k8s kümmert sich um das Loadbalancing.

Service-Discovery ist somit bereits vorhanden.

Ein „Service“ im Kontext von Kubernetes meint also nicht einen SpringBoot-Service, letzteres ist eine Application.

```
apiVersion: v1
kind: Service
metadata:
  name: www
spec:
  ports:
  - port: 8080
    targetPort: 8080
  selector:
    app: www
```

Networking API-Objekte: Ingress / Egress

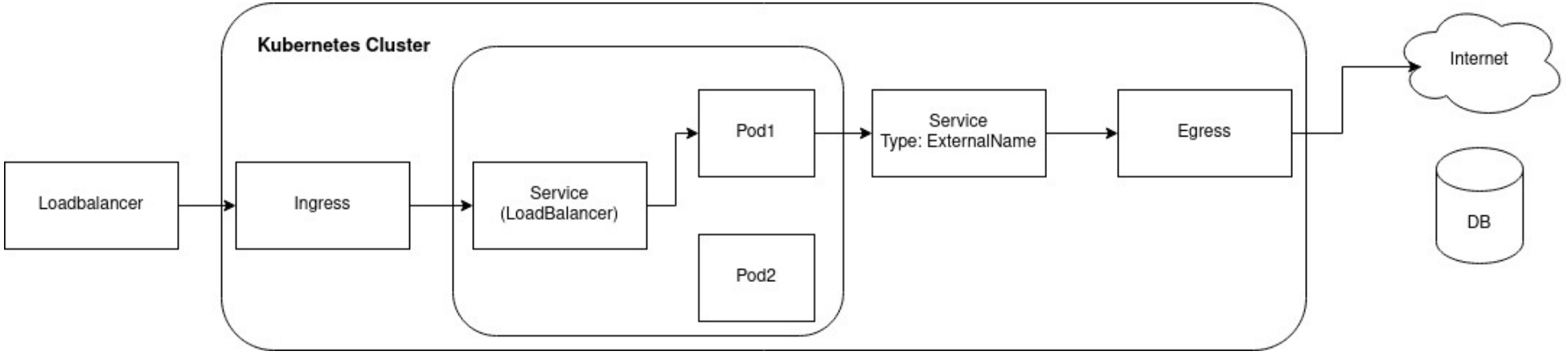
Ingress: Ein Network-Policy API-Objekt, das den externen Zugriff auf die Dienste in einem Cluster verwaltet, normalerweise HTTP(s).

Das Traffic-Routing wird durch Regeln gesteuert, die auf der Ingress-Ressource definiert sind.

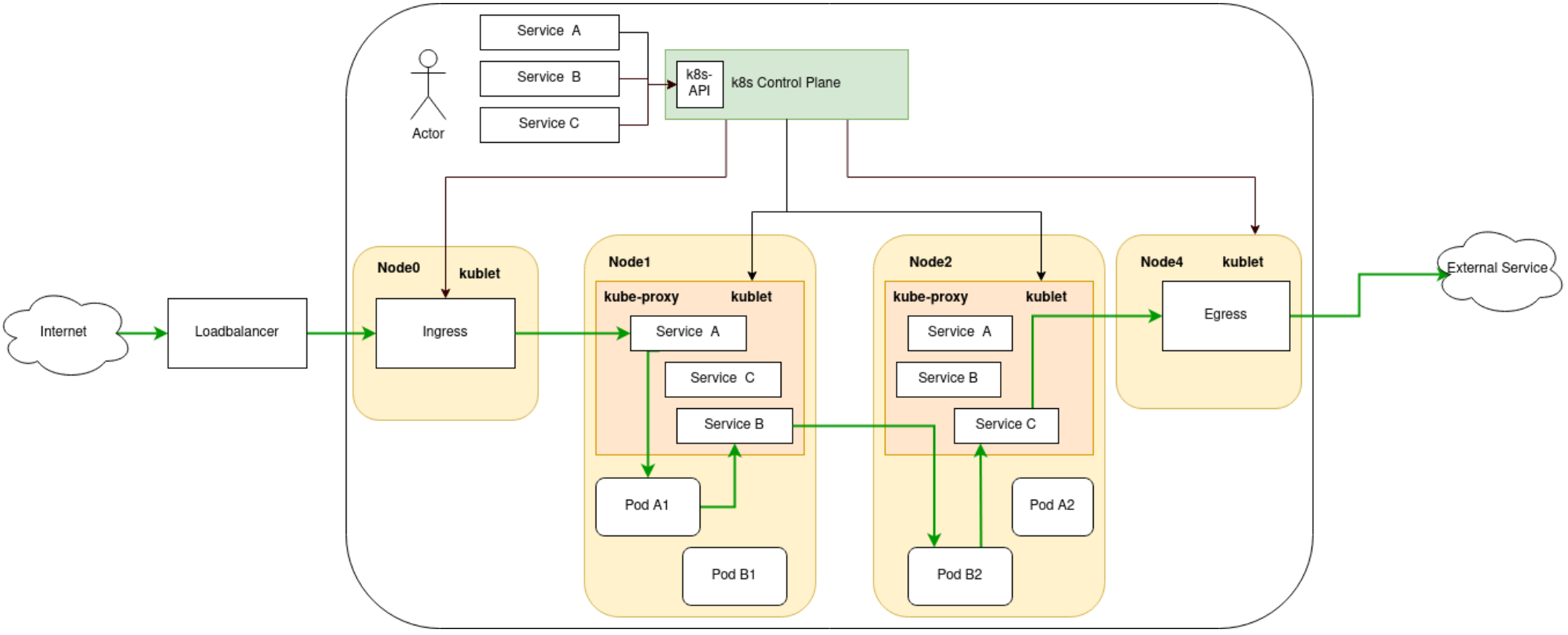
Ingress kann **Loadbalancing**, **SSL-Terminierung**

Egress: Ein Network-Policy API-Objekt, dass den ausgehenden Traffic eines Pods behandelt

k8s Netzwerktraffic



K8s Networktraffic



Exkurs: Pod SideCars

Weiterer Container der neben einem Container im Pod installiert werden kann um zusätzliche Aufgaben zu übernehmen. (Augmentierung)

Beispiele:

- Logging Sidecar, wenn man speziell auf Pod Ebene loggen möchte,
- NetzwerkProxy für die Interception der Netzwerkkommunikation eines Pods beim Einsatz von Istio Service Mesh.

k8s Networking - Istio

Istio ist eine Architekturschicht im Kubernetes Cluster zur Verwaltung des Netzwerktraffics

Istio ist dabei nicht nur **ServiceMesh** (ServiceDiscovery und Loadbalancing) sondern integriert auch Logging & Telemetrie des Netzwerktraffics sowie Netzwerksicherheit durch Zugriffsrichtlinien.

Die Istio-Architektur unterscheidet sich in:

- Datenlayer: **Envoy Proxies** an den Pods der WorkerNodes zur Steuerung der Netzwerkkommunikation
- Steuerungslayer: die **istiod** Control Plane erstellt die Konfiguration der Envoy Proxies für das Routing des Datenverkehrs anhand von API-Objekten wie Gateways, VirtualService, ServiceEntry

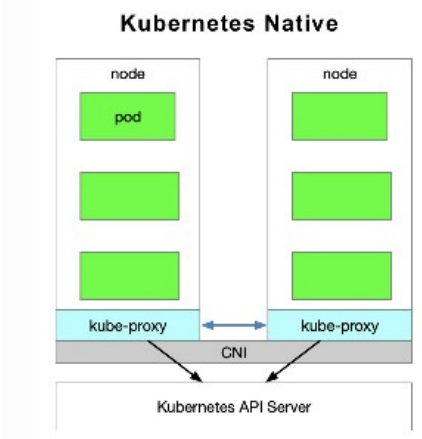
Istio -Service Mesh, Envoy

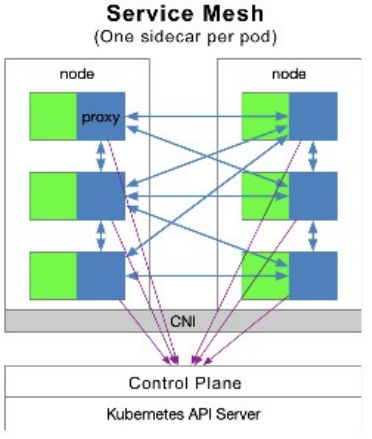
Istio ersetzt den nativen kube-proxy, weil der nur rudimentäre Funktionalität bietet.

Envoy ist der Istio-Agent am Pod und als SideCar am Pod installiert. Er fungiert als Netzwerk-Proxies und verbessert die Netzwerkfunktionalität eines Pods, sodass Pod zu Pod Kommunikation möglich ist, was die Netzwerk-Latency gegenüber der kube-proxy Variante verbessert. Außerdem kann kube-proxy nur global konfiguriert sein und nicht per Pod.

Dass der Netzwerk-Traffic durch den Envoy SideCar-Proxy geleitet wird, wir über einen Istio-InitContainer erreicht, der beim Start des Pods zunächst die entsprechen IP-Tables initialisiert.

Der ServiceMesh kann über kiali visualisiert werden

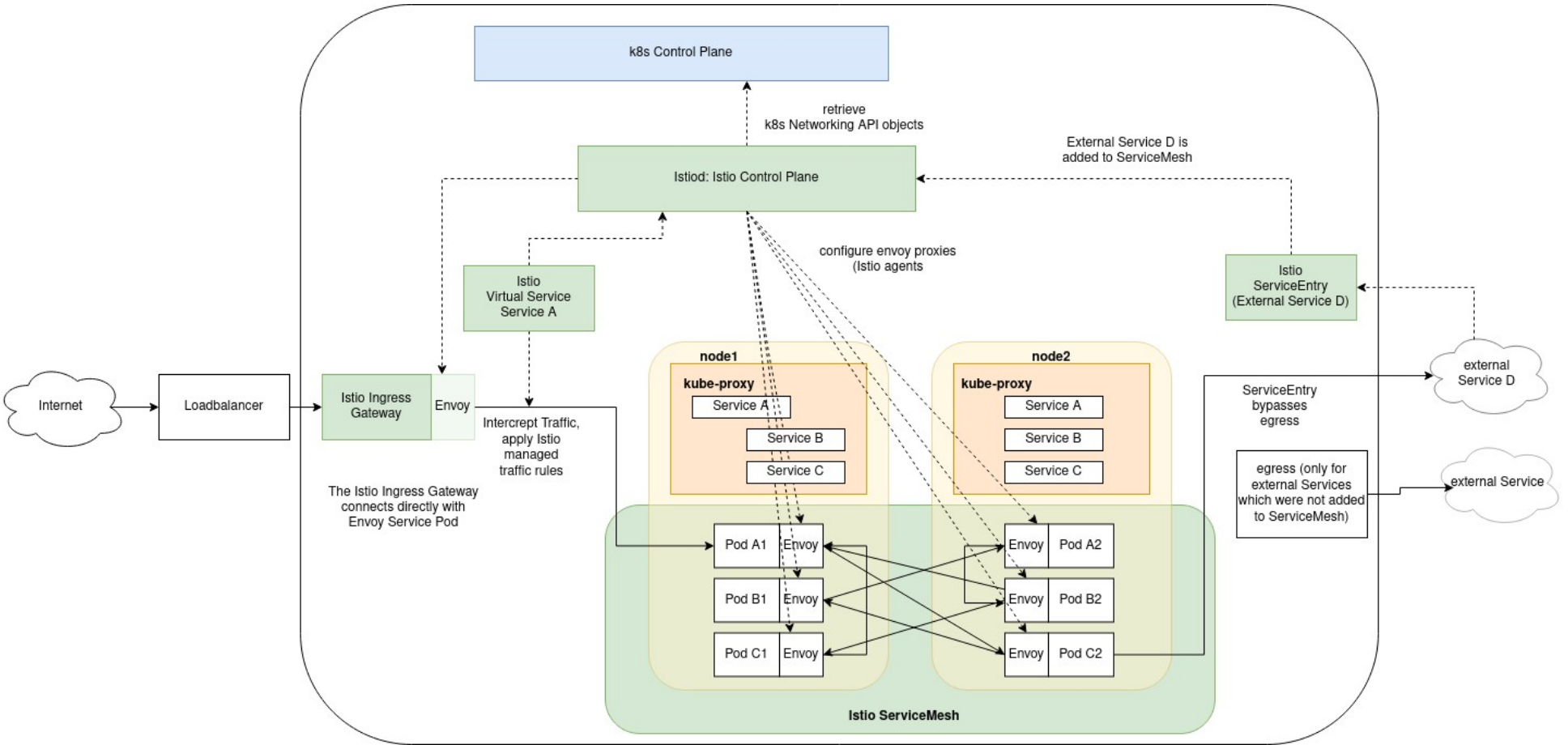




Istio Ingress, VirtualService, ServiceEntry

- **Istio Ingress Gateway** ersetzt den Kubernetes Ingress und leitet den Netzwerktraffic in den ServiceMesh, das Gateway kann direkt mit den Envoy-Proxies am Pod kommunizieren und umgeht somit den kube-proxy
- Ein **VirtualService** ist ein Istio k8s API-Objekt, das die Regeln für das Routing des Envoy-Traffics innerhalb des ServiceMesh definiert
  - Die VirtualServices werden über das helm Deployment angelegt
- Ein **ServiceEntry** bringt einen externen Service in den ServiceMesh, der ServiceEntry öffnet die Firewall und externe Services können somit von den Envoy-Proxies direkt angesprochen werden (und benötigen keinen egress)
  - Bsp: RabbitMQ ServiceEntry im test-eks Cluster

k8s Netzwerktraffic mit Istio



Networking – Service Calls

Aufruf eines Service von außerhalb des Clusters:

- <namespace>-<service-name>.<cluster>.local

Aufruf eines Service innerhalb des Clusters:

- pod-to-service-Kommunikation
- <service-name>.<namespace>.svc.cluster.local

ExternalName Service:

- Aufruf eines externen Service der noch auf dem DockerHosts deployt ist

K8s Authorisierung: RBAC

- **RBAC** = role-based-access
- **ABAC** = account-based-access
- Kubernetes kennt **Rollen** und **ClusterRollen** die ein Set von Berechtigungen für eine bestimmte Kubernetes Resouce beinhalten.
- Rollen setzen die Berechtigungen immer innerhalb eines bestimmten Namespaces, ClusterRoles gelten für den gesamten Cluster.
- Über **RoleBinding** und **ClusterRoleBinding** werden diese Berechtigungen an ein bestimmtes Set an Usern, Gruppen oder **ServiceAccounts** zugewiesen.

Kubernetes API und RBAC

Weil das alles **REST** ist: jede Anfrage wird als (API-Group)**Resource**-Request gesehen der per HTTP kommt mit den entsprechen **HTTP-Verben**

Diese Verben werden dann an die Rollen oder Cluster-Rollen gebunden

HTTP verb	request verb
POST	create
GET, HEAD	get (for individual resources), list (for collections, including full object content), watch (for watching an individual resource or collection of resources)
PUT	update
PATCH	patch
DELETE	delete (for individual resources), deletecollection (for collections)

API-Objekte: Role, RoleBinding

Rollen und ClusterRollen und die RoleBindings sind natürlich auch alles API-Objekte und werden deklarativ per .yaml File definiert

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

```yaml
apiVersion: rbac.authorization.k8s.io/v1
# This role binding allows "jane" to read pod
# You need to already have a Role named "pod-
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
# You can specify more than one "subject"
- kind: User
  name: jane # "name" is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  # "roleRef" specifies the binding to a Role
  kind: Role #this must be Role or ClusterRol
  name: pod-reader # this must match the name
  apiGroup: rbac.authorization.k8s.io
```

k8s Resourcen-Planung

Deployments im Kubernetes verbrauchen Resourcen (CPU und RAM).

AWS eks und der Cluster im Cronon kosten Geld.

IT/OPS brauch eine Grundlage, anhand der die Größe des Clusters und die Ausstattung der WorkNodes kalkuliert werden kann.

Jeder Bounded Context kann nicht unbegrenzt Resourcen in Anspruch nehmen.

Resource-Limits werden pro BoundedContext über deren Kubernetes Namespaces per ResourceQuotas festgelegt. Die Einhaltung dieser ResourceQuotas wird von Kubernetes selbst überwacht.

Vorteile k8s - Skalierung

Skalierung ist über einen einzigen Eintrag in einer Konfiguration änderbar.

Kubernetes sorgt selbst dafür, dass die entsprechende Anzahl der Instanzen läuft und die Requests über den Loadbalancer im Kubernetes gleichmäßig verteilt werden.

Kubernetes kennt auch Optionen für eine automatische Skalierung und kann selbständig einen weiteren Container starten, wenn z. B. die Anzahl der Requests an diesen Service zunimmt und später auch wieder automatisch beenden.

Natürlich müssen innerhalb des Namespaces genügend Ressourcen dafür zur Verfügung stehen.

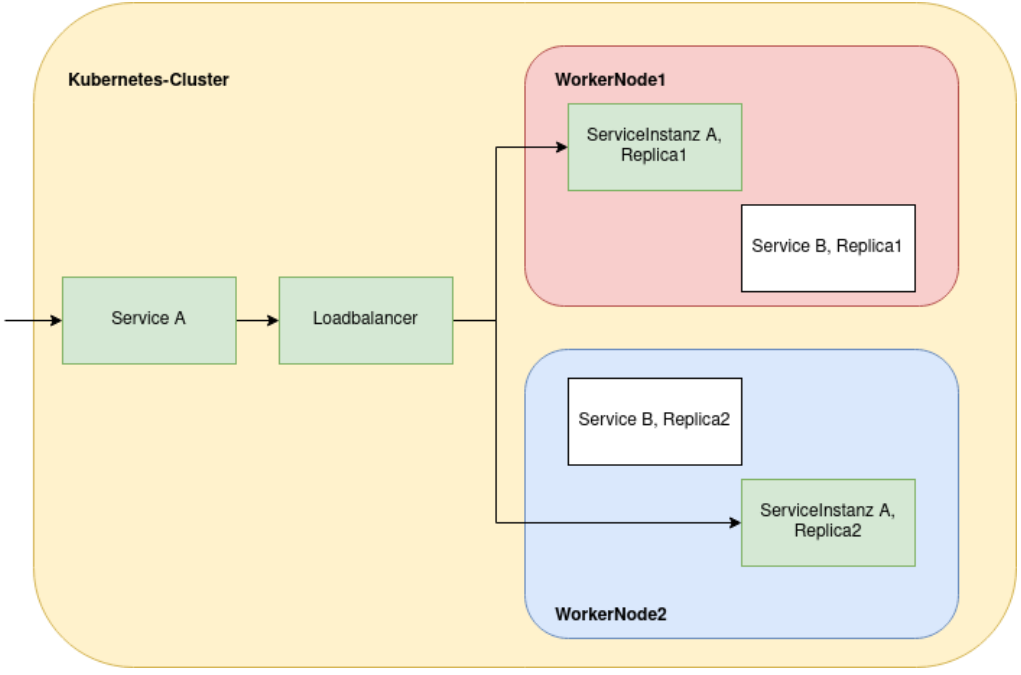
Möglichkeit über **knative** serverless Anwendungen zu betreiben

```yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
  name: php-apache
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: php-apache
  minReplicas: 1
  maxReplicas: 10
  targetCPUUtilizationPercentage: 50
```

Vorteile k8s - Ausfallsicherheit

Kubernetes versucht automatisch die Instanzen einer Anwendung auf Pods in verschiedenen WorkerNodes zu verteilen.

Wenn ein WorkerNode vom Netz geht wird die Anwendung trotzdem weiterlaufen.



k8s Deployment-Strategien

- *recreate*: terminate the old version and release the new one,
- *ramped*: release a new version on a rolling update fashion, one after the other,
- *blue/green*: release a new version alongside the old version then switch traffic,
- *canary*: release a new version to a subset of users, then proceed to a full rollout
- *a/b testing*: release a new version to a subset of users in a precise way (HTTP headers, cookie, weight, etc.).

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
  labels:
    app: nginx
spec:
  replicas: 3
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1
  selector:
    matchLabels:
```

Deployment im k8s - helm

Helm ist ein **package-manager** für Kubernetes und hilft dabei Anwendungen in Kubernetes zu deployen.

Dabei verwendet helm ein eigenes Templating, die Konfigurationsdateien aller benötigten Kubernetes API Objekte für ein Deployment, entsprechend dem jeweiligen Kubernetes Cluster (test,ref,prod - aws, cronon) zu generieren.

Wenn wir ein MicroService deployen (Giltlab-Projekt), dann verwenden wird das HighLevel Deployment API-Objekt.

Im Projekt selbst haben wir nur die Werte für die Erzeugung der ConfigMap in den values.yaml angelegt und die Secrets gesetzt.

Die Gitlab Pipeline erzeugt zunächst im **build**-Step ein Docker-Image der Anwendung und speichert das in der **Docker-Registry.**

Im **deploy** Step wird helm aufgerufen, um zunächst die Definitionen für die benötigten Kubernetes-API Objekte aus den helm Charts im Projekt generieren.

Der Name des zuvor erzeugten Docker-Images wird dabei auch als Value-Parameter an das helm Templating übergeben, um den Konfiguartionswert für „image“ im Deployment-Objekt zu setzen

Zuletzt erzeugt helm mittels **kubectl** dann die eigentlichen Kubernetes-API Objekte auf dem Kubernetes-Cluster.

Das im Deployment gesetzte Docker-Image zieht sich Kubernetes dann selbst aus der Docker-Registry

helm – Templates im Projekt

Beispiel:

- **configmap.yaml** – öffentliche Konfiguration Parameter der (SpringBoot)Anwendung, gesetzt stage-abhängig in den values/config.yam,
- **deployment.yaml** – Deployment Konfiguration der Anwendung,
- **secrets.yaml** - geheime Konfiguration Parameter der (SpringBoot)Anwendung, gesetzt stage-abhängig in den values/secrets.yaml,
- **service.yaml** – Netzwerk-Service Interface für alle Pods in denen die Anwendung läuft,
- **serviceaccount.yaml** – Wenn die Anwendung spezielle Permissions braucht können die über RoleBindings diesem Account zugeordnet werden (RBAC),
- **virtualservice.yaml** – Damit kann der Netzwerk-Traffic der Anwendung über Istio verwaltet werden.

Diese Templates werden initial über das helm-starter in dem jeweiligen Projekt angelegt. Dieser helm-starter entspricht weitestgehend dem Standard, das auch bei dem Befehl helm create erstellt wird – mit GVL-spezifischen Ergänzungen.

Deployment im k8s - Argo CD

Über helm deployen wir unsere Projekte im Kubernetes (Deployment, Cronjob)

Über ArgoCD werden die Kubernetes API-Objekte deployt, die für die Infrastruktur des Clusters verantwortlich sind.

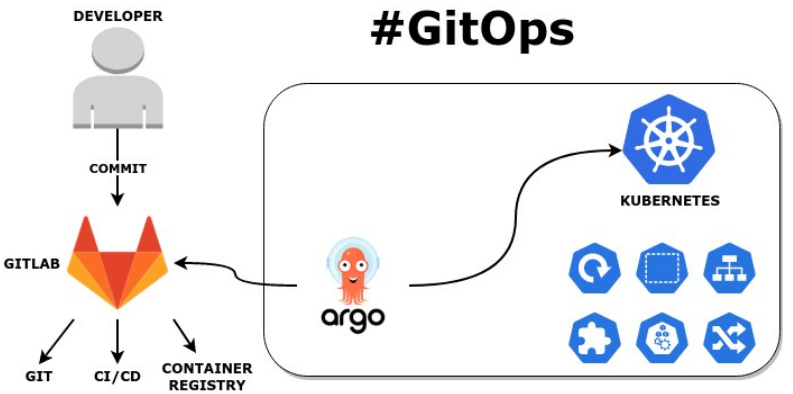
Deklaratives GitOps CD (ContentDelivery) für Kubernetes.

Argo CD ist Kubernetes native, es ist ein Kubernetes Controller, der den Cluster überwacht und fortlaufend den Zustand des Clusters mit dem im Git eingecheckten erwarteten Zustand abgleicht. Somit wird jede Änderung der Kubernetes Konfiguration durch einen Git Commit sofort in den Cluster übertragen.

ArgoCD nutzt kustomize, eine Overlay-Engine als kubectl plugin, für Environment-spezifsches Deployment

Über ArgoCD wird bspw verwaltet:

- Ingress, Egress
- Namespaces,
- Fluentd,
- RBAC,
- Istio, Istio API Objekte



fluentd – Logging - Graylog

Wie kommen die Logs ins Graylog?

Die Applikationen in den Pods loggen alle per Standard.out/err in var/log des WorkerNodes.

fluentd Appikationen werden per DaemonSet auf jedem WorkerNode installiert .

fluentd hat var/log gemounted und bekommt so die Logs aller Pods/Applikationen des WorkerNodes und sendet sie an Graylog.

Ein DaemonSet ist eine weitere Workload-Resource, die sicherstellt, dass eine Kopie dieses fluentd-Pods auf jedem WorkerNode im Cluster läuft.

Lens – Kubernetes IDE

UI um mit dem Kubernetes Cluster zu interagieren.

Es nutzt die Kubernetes-API.

In Abhängigkeit von den Berechtigungen des Users werden:

- die Kubernetes Objekte im Cluster angezeigt,
- besteht die Möglichkeit diese Objekte über die UI zu ändern.

  Bspw.:

  - Deployments stoppen,
  - Cronjobs starten.

kubectl

- cli Tool für Kubernetes
- kubectl cheat sheet
  - kubectl config current-context
  - kubectl -n rdx get deployments
  - kubectl n rdx describe <deploymentname>
  - kubectl apply -f <deployment.yaml>

IT/OPS vs Developer vs DevOPS

IT/OPS

- Aufsetzen des Kubernetes Clusters, Sicherstellen des Betriebs,
- hinzufügen von weiteren WorkerNodes,
- sicherstellen, dass der Cluster mit der immer selben Konfiguration läuft und wiederhergestellt werden kann,
- kennt Cloud-Anbieter, Cloud-Spezifika,
- DNS und Netzwerkzonen einrichten,
- Einbindung der externen Komponenten S3, PostgresDB, Mongo-Atlas, Graylog, Prometheus,
- Netzwerksicherheit, und vieles mehr …
- **GIT-OPS**: Praktik zum Management von Infrastruktur- und Anwendungskonfiguartionen mithilfe von Git. Das Git Repository ist die „**Single Source of Truth**“ für das deklarative Management. Der gesamte Zustand des System ist im Git gespeichert.
- Tools:
  - **Terraform** – Aufsetzen des Kubernetes Clusters (cloud-spezifisch),
  - **ArgoCD**.

Developer

Deployen die MicroServices in Kubernetes durch Anlegen der Kubernetes Objekte über die Gitlab Pipeline.

Es können nur Kubernetes Objekte innerhalb des Kubernetes Namespace des Bounded Contexts angelegt werden.

Es werden nur Workload Resources angelegt.

Keine Netzwerk-Konfiguration.

Kein RBAC.

DevOPS

Zusammenführung der beiden Teilbereiche Development (Dev) und IT-Operations (Ops).

Verständnis für:

- Konfiguration des Clusters um den Betrieb der Applikationen sicherzustellen,
- Netzwerke,
- Cluster-Monitoring,
- Richtet die Kubernetes Objekte außerhalb der Namespaces der BoundedContexte ein:
  - Istio,
  - Fluentd,
  - RBAC, ...
- Festlegung der ResourceQuotas per Namespace (BoundedContext),
- und natürlich Entwicklung.

Hat Commit Rechte auf das Kubernetes Konfigurations-Repo im Gitlab (pltf/kubernetes)