Überblick über die Architektur von Kubernetes

Anatoli Kreyman

Einführung

Das Buch "Überblick über die Architektur von Kubernetes" ist ein umfassender Leitfaden, der ein tiefes Verständnis der Architektur und Funktionsweise von Kubernetes vermitteln soll.

Das vermittelte Wissen und ein Minimum an praktischer Erfahrung sind vollkommen ausreichend, um die Kubernetes Basiszertifizierung (Kubernetes and Cloud Native Associate) zu erlangen.

Bildnachweise:

Alle hier verwendeten Diagramme wurden von mir selbst erstellt und können von Ihnen verwendet werden.

Inhaltsverzeichnis

Einführung 2

Was ist Kubernetes? 8

VM vs. Container oder OS Isolierung vs. Applikation Isolierung 8

Microservices 8

Verwendungsszenarien 9

Einsatzszenarien 9

Kubernetes Vorteile 9

Flexibilität 9

Skalierbarkeit / Effizienz 9

Ausfallsicherheit 9

Deklarative Konfiguration 9

Self-Healing 10

Ökosystems 10

Kubernetes Nachteile 10

Kosten und Komplexität 10

Nicht immer sinnvoll 10

Kubernetes-Architektur Diagramm 11

Control Plane 12

Kubernetes API-Server 12

ETCD 12

kube-controller-manager 12

kube-scheduler 13

Worker Nodes 13

kubelet 13

Kube-proxy 13

IP-Tables-Modus 14

IPVS-Modus 14

Endpunkt-Modus 14

Container Runtime 14

API Server 15

RESTful API 15

kubectl api-resources 16

Authentifizierung, Autorisierung, Validierung und Zulassung 16

Authentifizierung 16

Autorisierung 18

Validierung 18
Zulassung 18
Informationsaustausch mit ETCD 18
Lesen von Daten 18
Schreiben von Daten 18
Beobachten von Änderungen 18
Skalierbarkeit, Erweiterbarkeit, Versionskontrolle 18
Skalierbarkeit 19
Erweiterbarkeit 19
Versionskontrolle 19
API Objects 19
API Groups 19
Core-API-Groups 20
Named API-Groups 20
API Resource Location – Beispiele 20
API Versioning 21
Alpha 21
Beta 21
Stable 21
HTTP-Antwortcodes vom API-Server 22
Namespaces 23
Isolierung 23
Ressourcenverwaltung 24
Zugriffssteuerung 24
Namenstrennung 25
Vordefinierten Namespaces 25
Labels 25
Beispiel eines Kubernetes Labels: 26
Annotations 27
Beispiel einer Kubernetes Annotations: 28
Labels vs. Annotations 28
Workload-Objekte 29
ReplicaSet 29
Deployment 31
RollingUpdate – Parameter 32
kubectl Befehle für das Deployment 32

DaemonSet ... 33
StatefulSet ... 34
Headless Service ... 34
Jobs ... 35
CronJobs ... 36
Healthcheck-Objekte ... 36
Liveness Probe ... 36
Readiness Probe ... 37
Startup Probe ... 37
Taints und Tolerations ... 38
Taints und Tolerations - technische Umsetzung ... 38
Taint-Optionen ... 39
Typische Anwendungsfälle ... 40
NodeSelector ... 41
Node Affinity ... 42
Pod Affinity / Pod Anti-Affinity ... 44
Pod Affinity ... 44
Pod Anti-Affinity ... 46
Kubernetes Netzwerk ... 47
CNI-Plugin ... 47
Service-Discovery ... 47
Network-Policies ... 47
Pod-to-Pod-Kommunikation ... 47
Netzwerk-Arten im Kubernetes Cluster ... 47
Node Network ... 48
Pod Network / Cluster Network ... 48
Service Network ... 49
Kubernetes Ingress ... 50
Name-based Virtual Hosts ... 50
Path-based Routing ... 50
TLS Termination ... 50
Service API vs. Ingress ... 50
Open Source Ingress Controller ... 51
Ingress Beispiel ... 51
Kubernetes Egress ... 51
Ergress Beispiel ... 52

Kubernetes DNS (kube-dns) 52
Kubernetes CoreDNS 52
Kubernetes Netzwerk-Plugins / Container Networking Interface (CNI) 53
Service-Typen 53
ClusterIP 54
Beispiel ClusterIP 54
NodePort 55
Beispiel NodePort 55
LoadBalancer 56
ExternalIPs 56
Beispiel ExternalName 57
NodePort vs. ExternalIPs 57
ExternalName 58
Beispiel ExternalName 58
Kubernetes Storage 59
Volumes 59
Persistent Volume 60
Persistent Volume Claim 61
PVC-Beispiel 61
Access Modes 61
Static Provisioning 62
Dynamic Provisioning 62
Storage Class 62
Storage Lifecycle 62
Autoscaling 63
Was ist Skalierung? 63
Was ist Kubernetes Autoscaling? 63
Autoscaling-Funktionen für Kubernetes 64
Vertical Pod Autoscaler (VPA) 64
Horizontal Pod Autoscaler (HPA) 65
HPA Komponenten 65
HPA-Versionen 66
Cluster Autoscaler 66
Requests und Limits 67
Requests 67
Limits 67

Einheiten ... 68
CPU-Throttling ... 68
Out-Of-Memory (OOM) ... 69
Resource Quotas ... 70
Quality of Service (QoS) ... 71
Guaranteed ... 71
Burstable ... 71
BestEffort ... 71
Secrets ... 72
Arten von Secrets ... 72
Speicherung der Secrets ... 72
Secret Encryption Config ... 72
Verwendung von Secrets-Management-Tools ... 73
Base64 Sicherheitsbedenken ... 73
KCNA – Prüfung ... 74
Inhalt der Zertifizierung (offizielle Information) ... 74
Kompetenzbereiche (offizielle Information) ... 74
Prüfungsdetails ... 74

Hinweis / Disclaimer

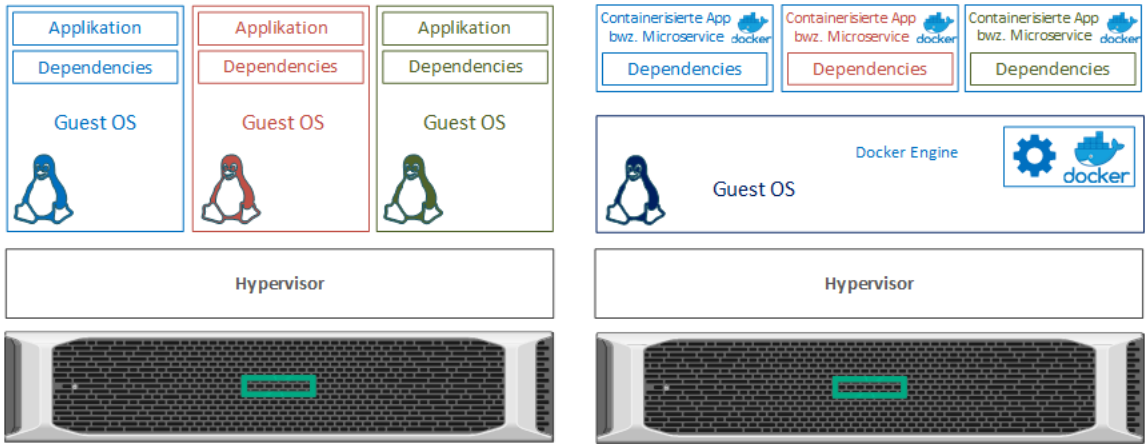
Dieses kleine Buch ist während meiner persönlichen Vorbereitung auf die Kubernetes and Cloud Native Associate (KCNA)-Prüfung entstanden. Es spiegelt meinen damaligen Wissensstand und mein Verständnis von Kubernetes wider. Ich kann daher weder die vollständige Korrektheit noch die Aktualität der Inhalte garantieren – insbesondere, da sich Kubernetes und das Cloud-Native-Umfeld ständig weiterentwickeln.

Die bereitgestellten Informationen dienen ausschließlich zu Lern- und Informationszwecken. Eine Haftung für eventuelle Fehler oder daraus entstehende Konsequenzen wird ausgeschlossen.

Was ist Kubernetes?

Kubernetes ist eine marktführende Orchestrierungsplattform für die Bereitstellung, Skalierung und Verwaltung von Containern in einem oder mehreren Clustern. Kubernetes (auch K8s genannt) wurde als Open-Source-Technologie im Jahr 2014 von Google ins Leben gerufen und wird von *Cloud Native Computing Foundation* weiter geführt. Inzwischen ist Kubernetes der De-facto-Standard für die Container-Orchestrierung geworden. Kubernetes wird oft mit „K8S“ abgekürzt — zwischen dem „K“ und dem „S“ stehen 8 Buchstaben.

VM vs. Container oder OS Isolierung vs. Applikation Isolierung



Zunächst ein paar Worte zur Containerisierung. Im Vergleich zu einer klassischen Applikation, die direkt auf einem Betriebssystem einer VM installiert wird, benötigen containerisierte Applikationen kein sehr abgespecktes Betriebssystem (z.B. Windows Core) und enthalten alle notwendigen Abhängigkeiten in einem Container selbst. Die Vorteile dieser Technologie sind:

- wesentlich weniger Hardware-Ressourcen werden benötigt (massive Kosten-Ersparnisse, besonders in der Cloud interessant)
- ein Container startet viel schneller als jedes Betriebssystem
- die Sicherheit wird durch einen kleineren Footprint erhöht
- die Software-Versionierung wird vereinfacht
- eine Plattform-Unabhängigkeit wird gewährleistet
- ein Container ist oft eine Basis für die Microservices-Architektur

Microservices

Microservices ist ein Begriff aus der Softwareentwicklung. Vereinfacht gesagt bedeuten Microservices nichts anderes als die Zerlegung einer großen (monolithischen) Anwendung in kleine Einzelteile, die bestimmte Dienste/Geschäftsfunktionen abbilden können.

In den letzten Jahren hat die Microservices-Architektur an Popularität gewonnen. Dies ist unter anderem auf die Verbreitung der Containerisierung zurückzuführen oder hat der Trend zur Microservice-Architektur die Containerisierung vorangetrieben (darüber streiten sich die Gelehrten). Wie jede Technologie hat auch die Microservice-Architektur Vor- und Nachteile.

Die gemeinsame Verwendung von Microservices und Kubernetes reduziert die möglichen Nachteile beim Übergang zur Microservices-Architektur.

Verwendungsszenarien

Einsatzszenarien

Je nachdem, mit wem man über Kubernetes spricht, wird man zwei Meinungen hören. Der Softwareentwickler sieht in dieser Technologie ein perfektes Werkzeug, um seine Produkte zu testen und damit die Softwareentwicklung zu beschleunigen bzw. Releasezyklen zu verkürzen. Aus Sicht des Managements können nun viele produktive Bereiche containerisiert und damit der Weg vom Test in die Produktion beschleunigt werden. Die Liste der Vorteile scheint mir also deutlich länger zu sein als die der Nachteile.

Kubernetes Vorteile

Flexibilität

Kubernetes passt perfekt zu einem weiteren Trend und ermöglicht eine „schmerzfreie" Migration in der Cloud. Alle großen Cloud-Anbieter (nicht nur AWS, GCP, Azure) bieten mittlerweile eine eigene Kubernetes-Infrastruktur an. Es ist auch möglich, mehrere Anbieter miteinander oder mit einer lokalen Infrastruktur zu verbinden.

Skalierbarkeit / Effizienz

Kubernetes automatisiert die horizontale Skalierung (Scale out) durch Hinzufügen oder Entfernen von Containern, basierend auf aktuellen Auslastungsindikatoren. Die automatische vertikale Skalierung (Scale up) sorgt für eine effiziente Zuteilung der im Cluster verfügbaren Hardwareressourcen. Die Verwendung von CI/CD- Pipeline (Continuous Integration / Continuous Delivery) ist eine sinnvolle Ergänzung zu einer Kubernetes-Infrastruktur und trägt zur Effizienz bei.

Ausfallsicherheit

Kubernetes beinhaltet einige Mechanismen, um die Hochverfügbarkeit sowohl für die Infrastruktur selbst als auch für die darauf laufenden Applikationen zu gewährleisten. An dieser Stelle ist anzumerken, dass die Applikation in der Lage sein muss, mit dem plötzlichen Ausfall von Containern umzugehen. Es wird zwar automatisch ein neuer Container gestartet, aber dessen Zustand bleibt nicht erhalten. Auch ein „VMotion" von Containern ist nicht vorgesehen – es wird immer ein neuer Container gestartet. Darüber hinaus überwacht Kubernetes kontinuierlich den aktuellen Zustand des Clusters. Hinzu kommen ein effizientes Traffic-Routing und Load Balancing.

Deklarative Konfiguration

Eine deklarative Beschreibung bzw. Konfiguration der Infrastruktur trägt ebenfalls zur Stabilität der Kubernetes-Infrastruktur bei. Im Gegensatz zu einer imperativen Konfiguration, die in Form von klaren Anweisungen agiert (wie z.B. Create instance A, Create instance B, Create instance C), definiert eine deklarative Konfiguration lediglich die Anzahl der benötigten Instanzen (z.B. Anzahl der laufenden Instanzen = 3). Diese Methode macht auch das Rollback zu einer früheren Version sehr einfach. Kubernetes wird immer versuchen, den in der deklarativen Beschreibung definierten Zustand herzustellen, aber es gibt keine Garantie, dass dieser Zustand erreicht wird.

Self-Healing

Die Selbstheilungsfähigkeit ist zweifellos eine der besten Eigenschaften von Kubernetes. Wenn eine containerisierte Anwendung oder ein (im Container platzierter) Pod abstürzt, startet Kubernetes die abgestürzten Komponenten neu (sofern genügend Ressourcen verfügbar sind und die Anwendung den Ausfall „verkraften“ kann).

Ökosystems

Ein Vorteil einer populären Open-Source-Lösung ist ein breites Ökosystem (Security, Monitoring-, Reporting- und Visualisierungs-Tools wie z.B. ElasticSearch + Kibana). Das hilft die Nutzbarkeit des Produktes zu verbessern. Auch in diesem Fall würden die hilfreichen Erweiterungen (wie z.B. Prometheus) in der Regel keine zusätzlichen Kosten verursachen.

Unter diesem Link finden Sie die aktuelle Landkarte der Projekte der Cloud Native Computing Foundation: https://cncf.landscape2.io

Kubernetes Nachteile

Kosten und Komplexität

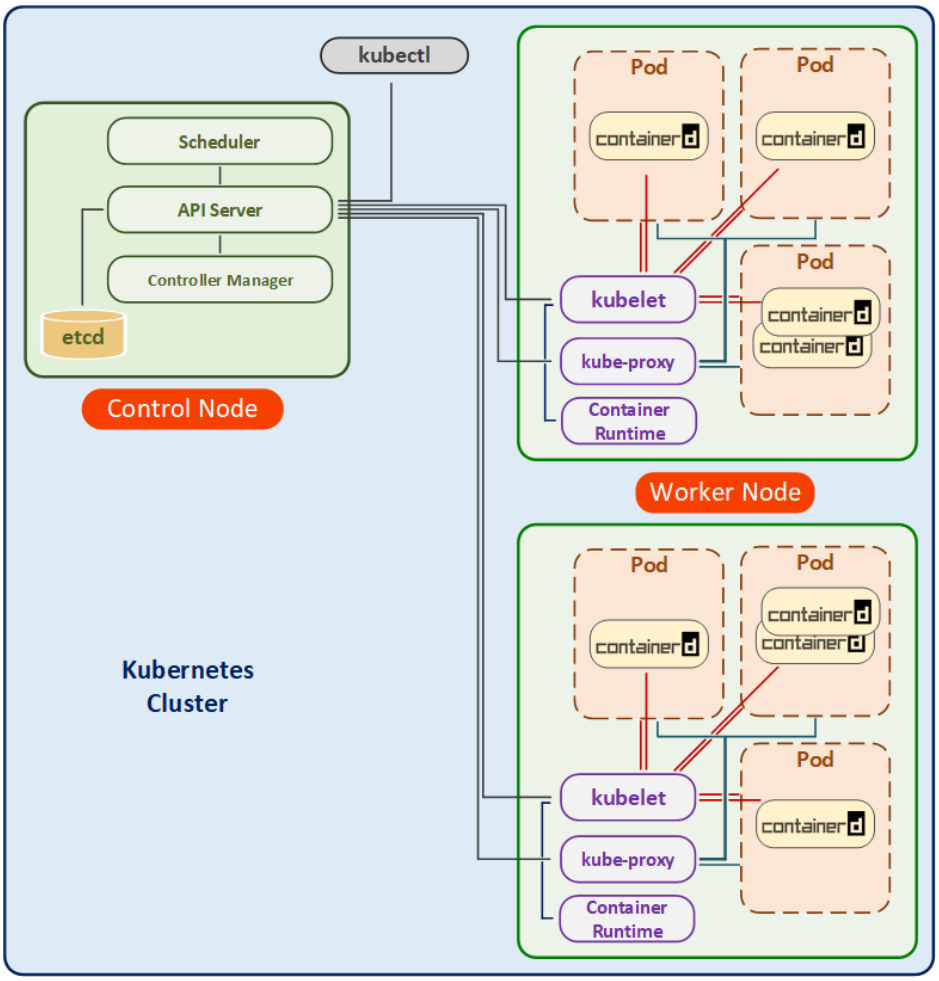
Die Umstellung auf Kubernetes kann teuer und umständlich sein. Selten kann man auf der grünen Wiese beginnen. In den meisten Fällen muss die bestehende Software so angepasst werden, dass sie problemlos auf Kubernetes läuft. Zunächst muss der finanzielle und zeitliche Aufwand für die Anpassung der Software abgeschätzt werden. Hierfür werden Experten mit fundierten K8s Kenntnissen sowie Softwareentwickler mit Erfahrung in der Entwicklung von containerisierten Anwendungen benötigt.

Nicht immer sinnvoll

Nicht für jede bestehende monolithische Anwendung ist es wirtschaftlich und/oder technisch sinnvoll, sie zu containerisieren. Auch die betrieblichen und organisatorischen Anpassungen können für manche Unternehmen und IT-Abteilungen eine größere Herausforderung darstellen.

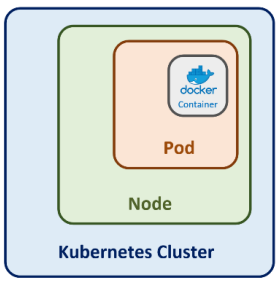
Kubernetes-Architektur Diagramm

Die Kubernetes-Architektur besteht im Groben aus zwei Schichten. Die erste Schicht könnte man als eine physikalische Schicht bezeichnen. Auf dieser Ebene befinden sich zwei Komponenten: ein oder mehrere *Master Nodes* sowie ein oder mehrere *Worker Nodes*.



Die weiteren Schichten sind zwei logische Abstraktionen:

- der Kubernetes Cluster selbst, welcher alle Komponenten beinhaltet
- die Pods, in denen einer oder mehrere Container Instanzen ausgeführt werden



Die Abbildung auf der rechten Seite verdeutlich die logische Gliederung der Komponenten: Cluster beinhaltet Nodes, Nodes beinhalten Pods, Pods beinhalten einzelnen Containers.

Configuration Maximums. Nicht mehr als ...	
110	Pods pro Node.
5000	Nodes.
150 000	Pods in einem Cluster.
300 000	Containers in einem Cluster.

Control Plane

Der Control Node ist eine wichtige Komponente von Kubernetes und besteht aus mehreren Bestandteilen, die eng miteinander interagieren, um den Kubernetes-Cluster zu steuern und zu verwalten.

Hier sind die wichtigsten Komponenten des Control Plane Nodes:

- **kube-apiserver**
- **etcd**
- **kube-controller-manager**
- **kube-scheduler**

Kubernetes API-Server

Der API-Server ist eine Komponente von Kubernetes, die es ermöglicht, den Cluster zentral zu verwalten und zu steuern. Durch die Verwendung der API können Entwickler und Administratoren Kubernetes-Anwendungen erstellen, die auf die Ressourcen des Clusters zugreifen und diese verwalten können.

Der Kubernetes-API-Server ist das primäre Gateway für die Interaktion mit dem Kubernetes-Cluster. Er bietet eine RESTful-API-Schnittstelle für die Verwaltung der Kubernetes-Ressourcen und ist für die Authentifizierung und Autorisierung von Benutzeranfragen verantwortlich.

ETCD

Kubernetes etcd wird verwendet, um Konfigurationsdaten und Informationen über den Zustand des Clusters, sowie über die Kubernetes-Ressourcen zu speichern und zu verwalten.

etcd ist eine zuverlässige, verteilte Datenbank, die von CoreOS entwickelt wurde und auf einfachen Schlüssel-Wert-Paaren basiert. Zwecks Hochverfügbarkeit und Skalierbarkeit kann etcd auf mehreren (auch separaten) Cluster Knoten ausgeführt werden.

etcd interagiert eng mit anderen Komponenten in der Architektur von Kubernetes. Der API Server greift auf etcd zu, um Informationen über Kubernetes-Ressourcen zu speichern und abzurufen. Der Kubernetes-Controller-Manager verwendet etcd, um Informationen über den Cluster-Status zu überwachen und automatisch den Zustand des Clusters anzupassen.

kube-controller-manager

Der Kubernetes kube-controller-manager ist eine Komponente von Kubernetes, die für die Überwachung und Verwaltung von Controller-Objekten im Cluster und die Durchsetzung des gewünschten Zustands (Desired State) verantwortlich ist. Der kube-controller-manager verwendet verschiedene Controller-Algorithmen, um Controller-Objekte zu verwalten. Zu diesen Algorithmen gehören der Replication-Controller, der Deployment-Controller, der StatefulSet-Controller und der DaemonSet-Controller.

kube-scheduler

Der Kubernetes kube-scheduler ist eine Architekturkomponente, die für die Zuweisung von Pods zu Nodes im Cluster verantwortlich ist. Der Scheduler wählt den geeigneten Node aus, auf dem ein Pod ausgeführt werden soll, basierend auf verschiedenen Faktoren wie den Ressourcenanforderungen des Pods, der Verfügbarkeit von Nodes und den spezifischen Anforderungen der Anwendungen. Der kube-scheduler verwendet einen Algorithmus, um den am besten geeigneten Knoten für die Ausführung des Pods zu finden. Der Algorithmus ist anpassbar und kann durch benutzerdefinierte Filter und Prioritäten erweitert werden.

Worker Nodes

Worker Nodes dienen der Ausführung von Pods. Ein Worker Node ist ein physischer oder virtueller Computer, auf dem eine Container-Laufzeitumgebung (z.B. containerd) ausgeführt wird. Ein Kubernetes-Cluster kann aus Hunderten oder Tausenden von Worker Nodes bestehen, je nach Größe des Clusters.

Ein Worker Node beinhaltet drei folgende Komponenten:

- **kubelet**
- **kube-proxy**
- **Container Runtime**

kubelet

Kubelet ist eine Komponente der Worker Node Architektur, die für die Verwaltung der Pods auf einem Worker Node im Cluster verantwortlich ist. Das Kubelet ist ein Agent, der auf jedem Node (kann auch auf den Control Nodes ausgeführt werden) in einem Kubernetes-Cluster läuft. Er ist verantwortlich für das Starten, Überwachen und Stoppen von Pods, basierend auf den Pod-Spezifikationen, die er vom Kubernetes API Server erhält.

Der Kubelet überwacht auch die Ressourcennutzung auf dem Node und stellt sicher, dass genügend Ressourcen für die laufenden Pods zur Verfügung stehen.

In Bezug auf das Netzwerk stellt er sicher, dass der Netzwerkstatus eines Pods korrekt gemeldet wird und dass der Pod die notwendigen Netzwerkressourcen erhält. Der Rest wird vom Kube-Proxy erledigt.

Kube-proxy

kube-proxy ist für das Routing des Netzwerkverkehrs innerhalb des Clusters zuständig. kube-proxy ermöglicht, dass die Anwendungen und Dienste innerhalb des Clusters über ihre Netzwerkadressen erreichbar sind, unabhängig davon, auf welchem Node sie ausgeführt werden.

kube-proxy verwendet verschiedene Modi, um den Netzwerkverkehr innerhalb des Clusters zu steuern. Die wichtigsten Funktionen und Eigenschaften sind hier kurz zusammengefasst:

IP-Tables-Modus

kube-proxy kann im IP-Tables-Modus betrieben werden, wobei das IP-Tables-Tool zur Definition und Verwaltung von Netzwerkregeln verwendet wird. Diese Regeln leiten den Datenverkehr je nach Konfiguration an die entsprechenden Pods weiter oder blockieren ihn.

IPVS-Modus

Im IPVS-Modus verwendet kube-proxy das IPVS-Modul (IP Virtual Server), das für fortschrittliches Load Balancing. IPVS bietet im Vergleich zum IP-Tables-Modus eine verbesserte Performance und Skalierbarkeit, insbesondere in großen und komplexen Clustern.

iptables-Modus

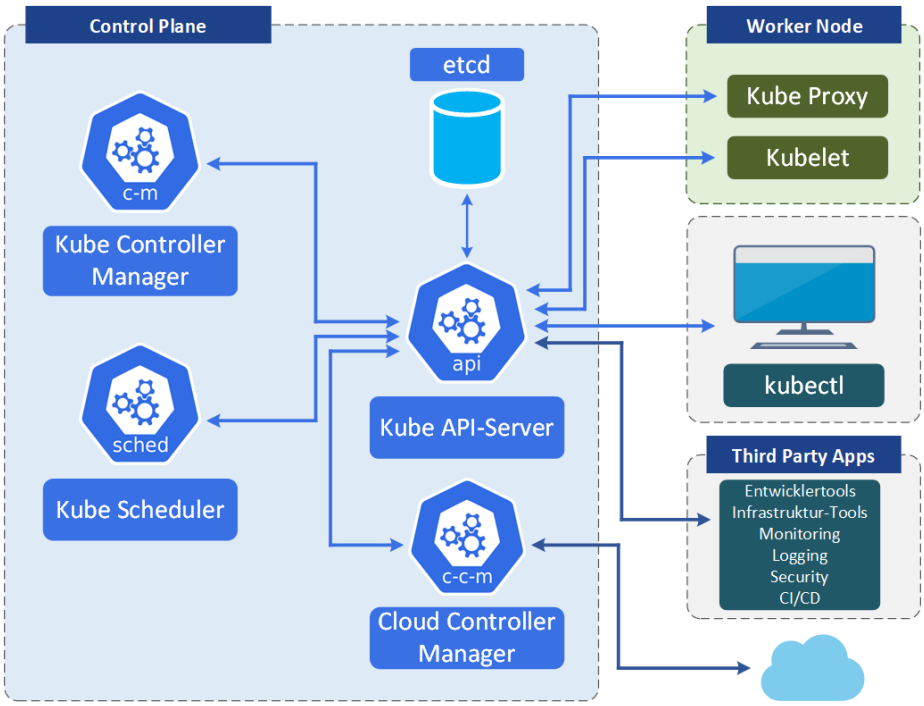
Im iptables-Modus programmiert Kube-Proxy iptables-Regeln auf jedem Node, um den Netzwerkverkehr von Services zu den zugehörigen Endpunkten (Pods) weiterzuleiten. Dabei werden die von Kubernetes verwalteten Endpoints bzw. Endpoint Slices verwendet, um eingehende Service-Anfragen per NAT an die passenden Pod-IP-Adressen zu verteilen.

Container Runtime

Die Container Runtime ist die dritte Komponente des Kubernetes Worker Nodes und ist für das Starten, Stoppen und Überwachen von Containern auf einem Worker Node verantwortlich. Sie stellt sicher, dass die Container gemäß den Spezifikationen der Kubernetes-Objekte (wie z.B. Pods) ausgeführt werden. Die zweite Aufgabe **Container-Images** aus einer Registry herunterladen und lokal bereitstellen.

API Server

Wie bereits erwähnt, ist der API-Server eine Komponente des Control Plane, der als zentrale Steuereinheit zwischen den verschiedenen Komponenten des Clusters und den Benutzern/Administratoren fungiert. Die Konfiguration und Steuerung des Kubernetes-Clusters ist nur über den API-Server möglich. Der Kubernetes API Server ist ein typisches Beispiel für eine Client-Server-Architektur.



Der API-Server selbst ist ein einzelner, in der Programmiersprache Go geschriebener Prozess. Der API-Server stellt eine RESTful API bereit, die für Authentifizierung, Autorisierung, Validierung, Zulassung, Ressourcenspeicherung, Informationsabruf sowie CRUD-Operationen (Create, Read, Update, Delete) verantwortlich ist.

Hier sind die Kernfunktionen der RESTful im erweiterten Überblick:

RESTful API

Im Allgemeinen ist eine RESTful API (Representational State Transfer) eine Softwarearchitektur, die es Systemen ermöglicht, über das Internet/Intranet per http miteinander zu kommunizieren und Ressourcen auszutauschen.

RESTful APIs basieren auf dem Konzept von Ressourcen. Eine Ressource ist ein Objekt (oft als Entität bezeichnet), auf das über eine eindeutige URL zugegriffen werden kann.

Der Client kann eine Anwendung, ein Skript oder ein System sein, das die API-Anfragen sendet. Der Kubernetes API-Server verarbeitet diese Anfragen. Dabei ist die RESTful API zustandslos, das bedeutet, dass keine Informationen über vorherige Anfragen speichert werden.

In einem Kubernetes-Cluster gibt es viele verschiedene Arten von Ressourcen, wie z.B. Pods, Deployments, Services und ConfigMaps. Jede dieser Ressourcen hat ihre eigene spezielle URL, über die man sie in der API aufrufen kann.

Die häufigsten http-Methoden in Kubernetes sind (die kursiv markierten Methoden gelten als speziell):

- GET - eine Ressource abrufen oder Informationen darüber erhalten
- POST - eine neue Ressource erstellen
- PUT - eine vorhandene Ressource aktualisieren oder ersetzen
- DELETE - eine vorhandene Ressource löschen
- PATCH - die angegebenen Felder einer Ressource ändern

Zusätzlich gibt es Kubernetes-spezifische Subresources/Operationen wie:

- *LOG - abrufen von Protokollen aus einem Container in einem Pod*
- *EXEC - ausführen eines Befehls in einem Container und Abrufen der Ausgabe*
- *WATCH - Änderungsbenachrichtigungen für eine Ressource mit Streaming-Ausgabe*

kubectl api-resources

kubectl api-resources - zeigt die Ressourcen zusammen mit ihren Kurznamen, API-Gruppen und ob sie in einem bestimmten Namespace verfügbar oder clusterweit verfügbar sind.

```
anatoli@admin-vm:~$ kubectl api-resources
NAME                              SHORTNAMES   APIVERSION                             NAMESPACED   KIND
bindings                                       v1                                     true         Binding
componentstatuses                 cs           v1                                     false        ComponentStatus
configmaps                        cm           v1                                     true         ConfigMap
endpoints                         ep           v1                                     true         Endpoints
events                            ev           v1                                     true         Event
limitranges                       limits       v1                                     true         LimitRange
namespaces                        ns           v1                                     false        Namespace
nodes                             no           v1                                     false        Node
persistentvolumeclaims            pvc          v1                                     true         PersistentVolumeClaim
persistentvolumes                 pv           v1                                     false        PersistentVolume
pods                              po           v1                                     true         Pod
podtemplates                                   v1                                     true         PodTemplate
replicationcontrollers            rc           v1                                     true         ReplicationController
resourcequotas                    quota        v1                                     true         ResourceQuota
secrets                                        v1                                     true         Secret
serviceaccounts                   sa           v1                                     true         ServiceAccount
services                          svc          v1                                     true         Service
challenges                                     acme.cert-manager.io/v1                true         Challenge
orders                                         acme.cert-manager.io/v1                true         Order
mutatingwebhookconfigurations                  admissionregistration.k8s.io/v1        false        MutatingWebhookConfiguration
validatingwebhookconfigurations                admissionregistration.k8s.io/v1        false        ValidatingWebhookConfiguration
customresourcedefinitions         crd,crds     apiextensions.k8s.io/v1                false        CustomResourceDefinition
apiservices                                    apiregistration.k8s.io/v1              false        APIService
controllerrevisions                            apps/v1                                true         ControllerRevision
daemonsets                        ds           apps/v1                                true         DaemonSet
deployments                       deploy       apps/v1                                true         Deployment
replicasets                       rs           apps/v1                                true         ReplicaSet
statefulsets                      sts          apps/v1                                true         StatefulSet
tokenreviews                                   authentication.k8s.io/v1               false        TokenReview
localsubjectaccessreviews                      authorization.k8s.io/v1                true         LocalSubjectAccessReview
selfsubjectaccessreviews                       authorization.k8s.io/v1                false        SelfSubjectAccessReview
selfsubjectrulesreviews                        authorization.k8s.io/v1                false        SelfSubjectRulesReview
subjectaccessreviews                           authorization.k8s.io/v1                false        SubjectAccessReview
horizontalpodautoscalers          hpa          autoscaling/v2                         true         HorizontalPodAutoscaler
cronjobs                          cj           batch/v1                               true         CronJob
jobs                                           batch/v1                               true         Job
certificaterequests               cr,crs       cert-manager.io/v1                     true         CertificateRequest
certificates                      cert,certs   cert-manager.io/v1                     true         Certificate
clusterissuers                                 cert-manager.io/v1                     false        ClusterIssuer
issuers                                        cert-manager.io/v1                     true         Issuer
certificatesigningrequests        csr          certificates.k8s.io/v1                 false        CertificateSigningRequest
```

Authentifizierung, Autorisierung, Validierung und Zulassung

Authentifizierung

Der API-Server ist verantwortlich für die Authentifizierung von Benutzern und Komponenten, die auf den Kubernetes Cluster zugreifen möchten. Der API-Server stellt sicher, dass nur authentifizierte Benutzer und Komponenten auf die Ressourcen zugreifen dürfen.

Autorisierung

Nach der Authentifizierung eines Benutzers oder einer Komponente prüft der API-Server, ob diese berechtigt sind, die angeforderte Aktion auf der angegebenen Ressource auszuführen. Der API-Server kann sowohl Role-Based Access Control (RBAC) als auch Attribute-Based Access Control (ABAC) verwenden, wobei RBAC die empfohlene und standardmäßige Methode ist.

Validierung

Der API-Server validiert eingehende API-Anfragen, nachdem diese authentifiziert und autorisiert wurden. Der API-Server prüft, ob die Daten in der Anfrage korrekt formatiert sind und alle erforderlichen Felder enthalten. Wenn die Anfrage die Validierung nicht besteht, wird sie zurückgewiesen und der Client erhält eine Fehlermeldung.

Zulassung

Nachdem eine Anfrage die Validierung bestanden hat, durchläuft er den Zulassungsprozess. Der API-Server kann auch zusätzliche Prüfungen durchführen und eventuell Anpassungen vornehmen, z.B. sind an dieser Stelle die "Admission Webhooks" (ValidatingAdmissionWebhook, MutatingAdmissionWebhook) integriert.

Anschließend werden die Anfragen, die authentifiziert, autorisiert, validiert und zugelassen wurden, die Ressourcen und ihr Status in der ETCD gespeichert.

Informationsaustausch mit ETCD

Die Kommunikation mit der Datenbank (ETCD) gehört ebenfalls zu der Kernaufgaben der API-Server. In diesem Fall ist der API-Server für diese Aufgaben verantwortlich: Lesen von Daten, Schreiben von Daten und Beobachten von Änderungen.

Lesen von Daten

Wenn eine Komponente (oder ein Benutzer über kubectl) den Zustand einer Ressource abfragen möchte, sendet sie eine Anfrage an den API-Server. Der API-Server liest dann die entsprechenden Daten aus ETCD und sendet sie an den Anfragenden zurück.

Schreiben von Daten

Wenn eine Komponente den Zustand einer Ressource ändern möchte, sendet sie eine Anfrage an den API-Server. Der API-Server validiert die Anfrage und schreibt die Änderungen in der ETCD.

Beobachten von Änderungen

Viele Komponenten in Kubernetes müssen auf Änderungen an bestimmten Ressourcen reagieren. Dies passiert, indem sie den API-Server auffordern, sie über Änderungen zu informieren. Der API Server hält eine Verbindung zu ETCD und wird über Änderungen informiert, die er danach an die beobachtenden Komponenten weiterleitet

Skalierbarkeit, Erweiterbarkeit, Versionskontrolle

Skalierbarkeit, Erweiterbarkeit und Versionskontrolle sind weitere Eigenschaften des API-Servers.

Skalierbarkeit

Die Architektur des API-Servers ermöglicht eine horizontale Skalierung, um sowohl die Leistung als auch die Verfügbarkeit zu erhöhen.

Erweiterbarkeit

Der API-Server ist erweiterbar und kann mit zusätzlichen Funktionen ausgestattet werden, ohne dass der Kern des API-Servers verändert werden muss. Die wohl bekannteste Möglichkeit ist die Verwendung von Custom Resource Definitions (CRDs). Die CRDs bieten die Möglichkeit, neue Ressourcentypen zu definieren, die der API-Server verstehen kann. Weitere Methoden sind Aggregated APIs, Admission Controllers, API Extensions und Webhooks.

Versionskontrolle

Der API-Server unterstützt die Versionskontrolle für die APIs, und ermöglicht damit die Abwärtskompatibilität. Mehr dazu unten im Abschnitt: API Groups

API Objects

Die API-Objekte sind die grundlegenden Einheiten (in der offiziellen Dokumentation als Persistent Entities bezeichnet), mit denen Kubernetes interagiert und die den Zustand des Kubernetes-Clusters repräsentieren.

Die API Objekte:

- repräsentieren und definieren den Zustand des Kubernetes-Clusters
- stellen alles dar, was in einem Kubernetes-Cluster existiert
- dienen als Basis, um den aktuellen Zustand mit dem gewünschten Zustand zu vergleichen
- dienen als Schnittstelle zwischen dem Benutzer und dem Kubernetes-System
- werden im YAML- oder JSON-Format beschrieben
- können über die Kubernetes API oder über kubectl erstellt, aktualisiert und gelöscht werden

Kubernetes API-Objekte sind auf drei Arten organisiert: *Kind*, *API-Group* und *API-Version*

Pod	DaemonSet	PersistentVolume	Node
Service	Job	PersistentVolumeClaim	Role
Deployment	CronJob	StorageClass	ClusterRole
ReplicaSet	ConfigMap	Ingress	RoleBinding
StatefulSet	Secret	Namespace	ClusterRoleBinding

API Groups

API-Gruppen ermöglichen eine bessere Organisation von Ressourcen in der Kubernetes-API, d.h. eine logische Strukturierung und Trennung voneinander.

Es gibt zwei Organisationsmethoden für API-Gruppen in Kubernetes: Core API-Gruppen und Named API-Gruppen.

Core-API-Groups

Die erste ist die Core API Group oder Legacy API Group. Diese Gruppe enthält Objekte, die zum Aufbau der grundlegendsten Ressourcen (wie Pods, Services und Nodes) verwendet werden. Als Kubernetes entwickelt wurde, gab es noch kein Konzept für API-Gruppen.

Named API-Groups

Mit der Weiterentwicklung von Kubernetes wuchs die Notwendigkeit, die neuen Objekte zu klassifizieren. So entstanden die „Named API Groups“. Ein typisches Beispiel für eine Named API Group finden Sie in der folgenden Tabelle. Sie werden auch feststellen, dass bei den neueren Named API Groups der Name der API Group auch Teil des URL Pfades wird.

API-Gruppe	API-Objekte
Core-API (v1)	Pod, Service, Volume, Namespace, Node, Event, Secret, ConfigMap, PersistentVolume, PersistentVolumeClaim
Named-API-Groups:	
apps	Deployment, DaemonSet, ReplicaSet, StatefulSet
batch	Job, CronJob
extensions	Ingress
rbac.authorization.k8s.io	Role, RoleBinding, ClusterRole, ClusterRoleBinding
admissionregistration.k8s.io	MutatingWebhookConfiguration, ValidatingWebhookConfiguration
apiextensions.k8s.io	CustomResourceDefinition
networking.k8s.io	NetworkPolicy
storage.k8s.io	StorageClass, VolumeAttachment

Um die API-Gruppen und ihre Versionen zu verwenden, müssen Sie den vollständigen Pfad einer API-Ressource angeben. Der Pfad setzt sich aus der API-Gruppe, der Version und der Ressource selbst zusammen. Zum Beispiel: */apis/apps/v1/deployments*

Weitere Information: https://kubernetes.io/docs/reference/kubernetes-api/

API Resource Location – Beispiele

Beispiele für URL-Pfade für Ressourcen in der Kubernetes-API:

Core API - URLs

- **Pod:** http://apiserver:port/api/v1/namespaces/{namespace}/pods/{pod-name}
- **Service:** http://apiserver:port/api/v1/namespaces/{namespace}/services/{service-name}
- **Volume:** http://apiserver:port/api/v1/persistentvolumes/{volume-name}

Named API Groups - URLs

- Deployment
  - Gruppe "apps"
  - http://apiserver:port/apis/apps/v1/namespaces/{namespace}/deployments/{deployment-name}
- NetworkPolicy

    - Gruppe "networking.k8s.io"
    - http://apiserver:port/apis/networking.k8s.io/v1/namespaces/{namespace}/networkpolicies/{networkpolicy-name}
- Role
    - Gruppe "rbac.authorization.k8s.io"
    - http://apiserver:port/apis/rbac.authorization.k8s.io/v1/namespaces/{namespace}/roles/{role-name}

Beschreibung:

{namespace} steht für den Namen des Namespaces
{pod-name}, {service-name}, {volume-name}, {deployment-name}, {networkpolicy-name} und {role-name} stehen für die Namen der Ressourcen

API Versioning

Die Kubernetes-API unterstützt verschiedene API-Versionen. Mehrere Versionen der Kubernetes-API können gleichzeitig auf einem Server vorhanden sein. Die API-Versionierung ermöglicht sowohl Abwärts- als auch Aufwärtskompatibilität, d.h. wir können die API-Version des Objekts, mit dem wir arbeiten wollen, in unserem YAML-Manifest angeben.

Während der Entwicklung durchläuft die API-Version drei Phasen des Entwicklungsprozesses: Alpha (V1alpha1) > Beta (V1beta1) > Stable (v1):

Alpha

Die Alpha-Version (experimental) ist die erste Entwicklungsstufe und enthält neue Funktionen, die sich noch in der Entwicklung befinden. Diese Funktionen können noch fehlerhaft sein und werden im Laufe der Zeit geändert oder entfernt.

Beta

Die Beta-Version (pre release) ist die zweite und stabilere Entwicklungsstufe als die Alpha-Version. In der Beta-Version wurden die Funktionen bereits getestet und verbessert. Änderungen an den Funktionen sind noch möglich.

Stable

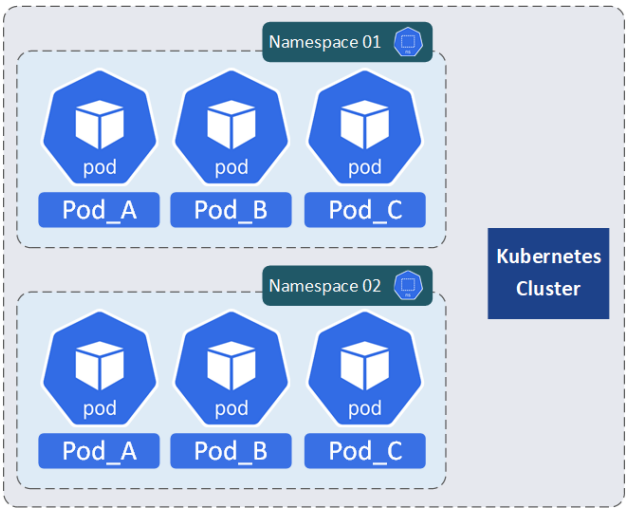
Die stabile Version (General Availability (GA)) ist die letzte Entwicklungsstufe und enthält ausführlich getestete Funktionen. In dieser Version gibt es in der Regel keine Änderungen mehr, sondern nur noch Fehlerbehebungen oder Sicherheitsupdates.

Die API-Versionen werden in der YAML-Konfigurationsdatei für Ressourcen und Workloads angegeben. Wenn eine Funktion als Alpha oder Beta gekennzeichnet ist, sollte sie verständlicherweise nur zu Entwicklungs- oder Testzwecken verwendet werden.

HTTP-Antwortcodes vom API-Server

HTTP-Statuscode	Bedeutung
200 OK	Die Anfrage war erfolgreich. Die Antwort des Servers enthält die angeforderten Daten.
201 Created	Die Anfrage war erfolgreich. Eine neue Ressource wurde erstellt.
202 Accepted	Die Anfrage wurde akzeptiert und wird verarbeitet. Die Verarbeitung ist noch nicht abgeschlossen.
204 No Content	Die Anfrage war erfolgreich. Kein Inhalt vom Server zurückgegeben.
400 Bad Request	Die Anfrage konnte aufgrund einer ungültigen Syntax nicht verstanden werden.
401 Unauthorized	Die Anfrage erfordert eine Benutzerauthentifizierung.
403 Forbidden	Der Server hat die Anfrage verstanden. Er weigert sich jedoch, sie auszuführen.
404 Not Found	Die angeforderte Ressource wurde auf dem Server nicht gefunden.
409 Conflict	Anfrage konnte aufgrund eines Konflikts mit dem aktuellen Ressourcenstatus nicht abgeschlossen werden.
500 Internal Server Error	Ein allgemeiner Fehler ist aufgetreten.

Namespaces



Namespaces sind sicherlich die wichtigste Methode, um Objekte in einem Kubernetes-Cluster in logische Einheiten zu organisieren. Die Verwendung von Namespaces ermöglicht die Aufteilung eines physischen Clusters in mehrere virtuelle Cluster. (Kubernetes-Namespaces hat nichts mit dem Konzept des Namespace des Linux-Betriebssystems zu tun)

Die Einsatzszenarien von Namespaces lassen sich grob in vier Bereiche unterteilen:

- Isolierung
- Ressourcenverwaltung
- Zugriffssteuerung
- Namenstrennung

Isolierung

Die Isolierung ist wahrscheinlich der Hauptgrund für die Verwendung von Namespaces in einem Kubernetes-Cluster. Die Isolierung bezieht sich sowohl auf die Sichtbarkeit als auch auf die Ressourcennutzung.

Alle in einem Namespace vorhandenen Ressourcen sind standardmäßig* nur innerhalb dieses Namespace sichtbar. Das bedeutet, dass z.B. Pods, Services, Volumes und andere Ressourcen, die in einem Namespace erstellt wurden, nicht direkt von einem anderen Namespace aus sichtbar oder zugänglich sind.

* Es gibt eine Reihe von Möglichkeiten, die Sichtbarkeit von Ressourcen zwischen Namespaces zu implementieren: Ingress Controller, Service Mesh, RBAC, Kubernetes Network Policies, Cluster-Scope Ressourcen. Nicht alle diese Tools und Konzepte haben direkt mit der "Sichtbarkeit" von Ressourcen zu tun haben, sondern eher mit Zugriffskontrolle, Routing und Kommunikation.

Ressourcenverwaltung

Aus technischer Sicht können Namespaces verwendet werden, um Ressourcenquoten zu definieren, die in einem Namespace verbraucht werden können. Durch die Verwendung von Namespace-basierten Ressourcenquoten kann verhindert werden, dass eine Anwendung alle Ressourcen des Clusters für sich beansprucht und dadurch andere Anwendungen beeinträchtigt.

Es gibt eine Vielzahl von Ressourcen, deren Verbrauch begrenzt werden kann. Hier sind einige Beispiele:

Ressourcenart	Begrenzt die < ... > die in einem Namespace erstellt oder beansprucht werden können.
pods	Gesamtzahl der Pods
services	Gesamtzahl der Services
persistentvolumeclaims	Gesamtzahl der PersistentVolumeClaims
secrets	Gesamtzahl der Secrets.
configmaps	Gesamtzahl der ConfigMaps
requests.cpu	Gesamtmenge an CPU-Zeit
requests.memory	Gesamtmenge an Speicher
limits.cpu	maximale Menge an CPU-Zeit
limits.memory	maximale Menge an Speicher
requests.ephemeral-storage	Gesamtmenge an temporärem Speicherplatz
limits.ephemeral-storage	maximale Menge an temporärem Speicherplatz

Für die technische Umsetzung ist der Kubernetes-Objekttyp (Kind) „ResourceQuota“ zuständig. Hier ist ein Beispiel wie die Anzahl der Pods in einem Namespace begrenzt wird:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: pod-quota
  namespace: mein-namespace
spec:
  hard:
    pods: "15"
```

Zugriffssteuerung

Die Namespaces übernehmen die Rolle der Sicherheitsgrenze für die rollenbasierte Zugriffskontrolle. Wir können auf der Basis von Namespaces einschränken, wer auf welche Ressourcen innerhalb eines Clusters zugreifen darf. Eine Ressource kann unter demselben Namen in mehreren Namespaces existieren. (RBAC kann sowohl auf Namespace-Ebene als auch auf Cluster-Ebene angewendet werden.)

Namenstrennung

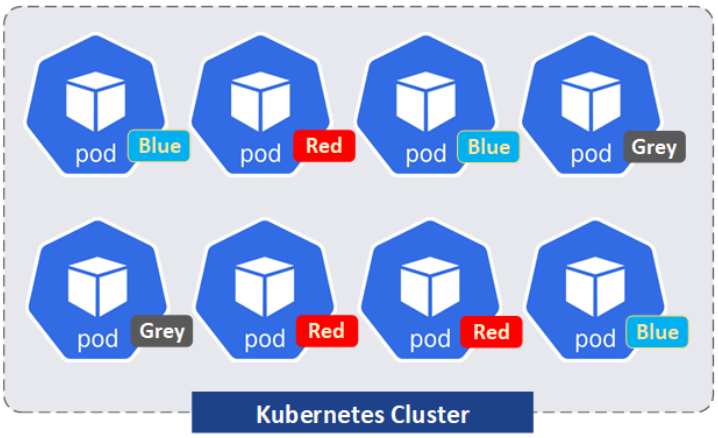
Die Namespaces können auch als eine Benennungsgrenze verwendet werden, wobei die Namen der Ressourcen nur innerhalb eines Namespace, nicht aber im gesamten Cluster eindeutig sein müssen.

Vordefinierten Namespaces

Nach der Installation eines neuen Kubernetes-Clusters stehen die folgenden vordefinierten Namespaces zur Verfügung:

Namespace	Beschreibung
default	Dies ist der Standard-Namespace, in dem Objekte erstellt werden, wenn kein anderer Namespace angegeben ist.
kube-system	Dieser Namespace ist für Objekte reserviert, die vom Kubernetes-System selbst erstellt werden.
kube-public	Dieser Namespace ist für Ressourcen vorgesehen, die für alle Benutzer öffentlich sichtbar und lesbar sein sollen.
kube-node-lease	Dieser Namespace enthält Lease-Objekte, die mit jedem Knoten verbunden sind.

Labels



Die zweite Methode, um Objekte / Ressourcen im Kubernetes-Cluster zu organisieren und zu kennzeichnen, wird als „Labels“ bezeichnet. Kubernetes Objekte/Ressourcen werden mit Labels versehen, um sie später leichter finden und auswählen zu können. Fast alle erstellbaren Ressourcen (Pods, Services, Volumes, Nodes, ReplicaSets, Deployments, StatefulSets usw.) können mit Labels versehen werden.

Im Großen und Ganzen werden die Labels in drei Szenarien verwendet:

- **Auswahl und Gruppierung**. Man könnte bestimmte Ressourcen anhand ihrer Labels auswählen und gruppieren.
- **Service Discovery**. Man könnte z.B. einen Service so konfigurieren, dass er nur die Pods verwendet, die entsprechend gelabelt sind.

- **ReplicaSets und Deployments**. In diesem Use Case werden nur die Pods von ReplicaSets / Deployments verwendet, die ein bestimmtes Label tragen.

Aus technischer Sicht sind die Labels Schlüssel-Wert-Paare (Key-Value Pair (KVP)). Die Schlüssel-Wert-Paare sind mit einer bestimmten Ressource in einem Cluster verknüpft (gelabelt).

- Der Schlüssel (key): der Schlüssel ist eine eindeutige Bezeichnung des Labels.
- Der Wert, der durch den Schlüssel repräsentiert wird.

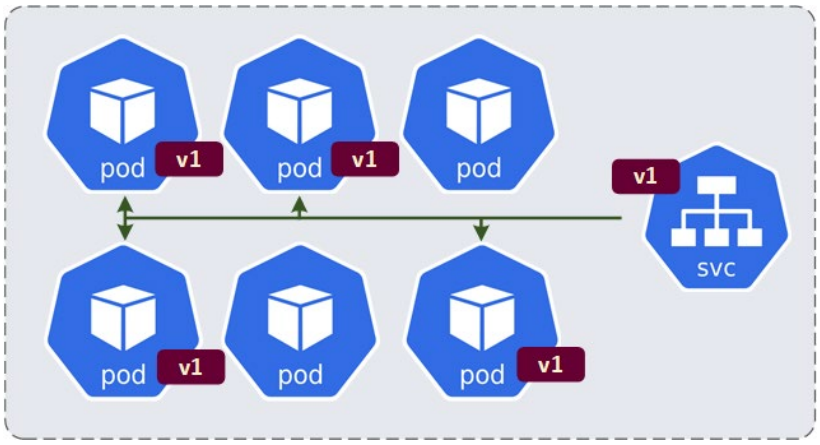
Beispiel eines Schlüssel-Wert-Paares:

- ***app=meineSuperApp***
- "***app***" ist der Schlüssel
- "***meineSuperApp***" ist der Wert.

Die Schlüssel-Wert-Paare können nicht beliebig benannt werden und unterliegen bestimmten Regeln. In Kubernetes dürfen die Schlüssel für Labels maximal 63 Zeichen lang sein und die Werte dürfen bis zu 253 Zeichen lang sein und dürfen die nur Buchstaben, Ziffern, Bindestriche und Unterstriche enthalten. Weitere Einzelheiten sind hier zu finden.

Beispiel eines Kubernetes Labels:

Im folgenden Beispiel wird ein Deployment (kind: Deployment) mit dem Namen "*mein-test-deployment*" erstellt.



Der *selector* im Abschnitt *spec* definiert, welche Pods von dem Deployment verwaltet werden. Der matchLabels definiert, dass das Deployment alle Pods auswählt, die das Label "*app=v1*" (blau markiert) haben.

Die labels innerhalb der *template* Spezifikation definieren die Labels, die an diese Pods angehängt werden. Die Labels "*app=v1*" (grün markiert) werden auf jedem Pod gesetzt, der auf der Grundlage dieser YAML-Datei erstellt werden.

Die Werte im *selector* (matchLabels: app: v1) sollen mit den Werten in den *labels* übereinstimmen bzw. miteinander gematcht sein.

apiVersion: apps/v1

```
kind: Deployment
metadata:
  name: mein-test-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: v1
  template:
    metadata:
      labels:
        app: v1
    spec:
      containers:
      - name: mein-supercontainer
        image: app-image
```

In der Service-Datei (kind: Service) dient der *selector* dazu, die Pods auszuwählen, an die der Service den Netzwerkverkehr weiterleitet soll. In diesem Fall sollte der „selector: app: v1“ mit den „labels: app: v1“ aus der Deployment-YAML gematcht werden.

```
apiVersion: v1
kind: Service
metadata:
  name: mein-service
spec:
  selector:
    app: v1
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
  type: ClusterIP
```

Annotations

Annotations ist die dritte Methode, um Objekte in Kubernetes zu organisieren. Annotations werden in der Regel von Benutzern verwendet, um Entscheidungen darüber zu treffen, was mit einer bestimmten Ressource auf der Grundlage der Annotationen geschehen soll. In den Regeln sind diese Informationen für die Verwaltung von Objekten durch Kubernetes nicht relevant, aber für bestimmte Werkzeuge (z.B. Build-, Release- oder Image-Informationen) können sie nützlich sein.

Die Verwendung von Annotations kann die Integration externer Datenquellen überflüssig machen, da alle notwendigen Daten bereits an die Ressource angehängt sind und sich innerhalb des Clusters befinden. Jeder Ressourcentyp in Kubernetes kann mit einer Annotation versehen werden.

Aus technischen Sicht sind die Annotations ebenfalls Schlüssel-Wert-Paare, die in der Sektion *Metadaten* beschrieben werden. Die Schlüssel von Annotations können bis zu 63 Zeichen lang sein, ähnlich wie bei Labels. Eine Annotation kann aber bis zu 256 KB lang sein und somit wesentlich mehr Informationen enthalten.

Beispiel einer Kubernetes Annotations:

So können die Annotations in einer YAML-Datei aussehen:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mydeployment
  annotations:
    last-checked: "2023-05-23T18:25:43.511Z"
    git-commit: "d4f0f834c0743264f0435f62f13c5e1e2899fb2"
    owner: Anatoli
    repository: "https://github.com/kubernetes/"
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mein-super-app
  template:
    metadata:
      labels:
        app: mein-super-app
    spec:
      containers:
      - name: mein-container
        image: mein-image
```

Labels vs. Annotations

Der Hauptunterschied zwischen Labels und Annotations liegt in ihrer Verwendung. Labels dienen der Identifizierung und Organisation von Kubernetes-Objekten und werden von Kubernetes selbst zur Verwaltung von Objekten verwendet. Annotations enthalten Informationen, die für die Verwaltung von Objekten durch Kubernetes nicht zwingend erforderlich sind, die aber für Entwickler, Operatoren und Tools nützlich sein können.

Workload-Objekte

Workload- und Ressourcenobjekte sind wichtige Bestandteile der Kubernetes-Architektur. Diese Objekte beschreiben, wie ein Container oder eine Gruppe von Containern in Kubernetes bereitgestellt und ausgeführt wird. Oft werden die Begriffe „Kubernetes Workload Objekte“ und „Kubernetes Controller“ als absolute Synonyme verwendet, obwohl dies nicht ganz korrekt ist.

Workload-Objekte sind API-Objekte für die Bereitstellung und Verwaltung von Anwendungen und Services in Kubernetes. Ein Workload-Objekt definiert einen bestimmten gewünschten Zustand und Kubernetes sorgt dafür, dass dieser Zustand aufrechterhalten wird.

Die Controller überwachen den aktuellen Zustand des Clusters und nehmen gegebenenfalls Änderungen vor, um den Cluster in den gewünschten Zustand (Desired State) zu bringen.

ReplicaSet

Ein ReplicaSet (auch ReplicaSet Controller genannt) sorgt für die Ausführung einer bestimmten Anzahl von Pods (Repliken eines Pods) in einem Cluster.

Die Begriffe *ReplicaSet* und *ReplicaSet Controller* werden oft synonym verwendet. Technisch gesehen handelt es sich um unterschiedliche Konzepte:

- Ein *ReplicaSet* ist ein Kubernetes-Objekt (YAML-Manifest). Es definiert, wie viele Kopien eines bestimmten Pods ausgeführt werden sollen.
- Der *ReplicaSet Controller* ist der Teil der Kubernetes-Kontrollebene, der dafür verantwortlich ist, dass die gewünschte Anzahl von Pods eines ReplicaSets erhalten bleibt.

Die Aufgabe des ReplicaSet Controllers ist es, zu erkennen, dass z.B. ein Pod (aus welchem Grund auch immer) beendet wurde und der Cluster vom gewünschten Zustand abweicht. In diesem Fall sollte der ReplicaSet Controller den fehlgeschlagenen Pod löschen und einen Create Request an den API Server senden, um einen neuen Pod im Cluster zu erzeugen und so den gewünschten Zustand wiederherzustellen.

In den seltensten Fällen werden ReplicaSets direkt erstellt. In der Regel werden sie durch Deployments erzeugt.

Die erforderlichen Elemente eines ReplicaSets sind:

- Pod Template
- Replicas
- Selector

Beispiel für eine ReplicaSet-Konfiguration:

```
apiVersion: apps/v1
```

```
kind: ReplicaSet
metadata:
  name: mein-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mein-app
    matchExpressions:
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: mein-app
        tier: frontend
    spec:
      containers:
      - name: mein-super-container
        image: mein-image
```

Erklärung:

- replicas - Anzahl der gleichzeitig laufenden Pods.
- selector - enthält die matchLabels und/oder matchExpressions
    - matchLabels - ist ein Schlüssel-Wert-Selektor . Alle von diesem ReplicaSet verwalteten Pods haben das Label *app* und den Wert *mein-app*.
    - matchExpressions - können zusätzlich oder anstelle von *matchLabels* verwendet werden und ermöglichen eine komplexere Logik bei der Auswahl von Pods.
- template - hier werden die Eigenschaften der Pods definiert.

Die matchExpressions-Komponente besteht immer aus drei Teilen: *key, operator* und *values.*

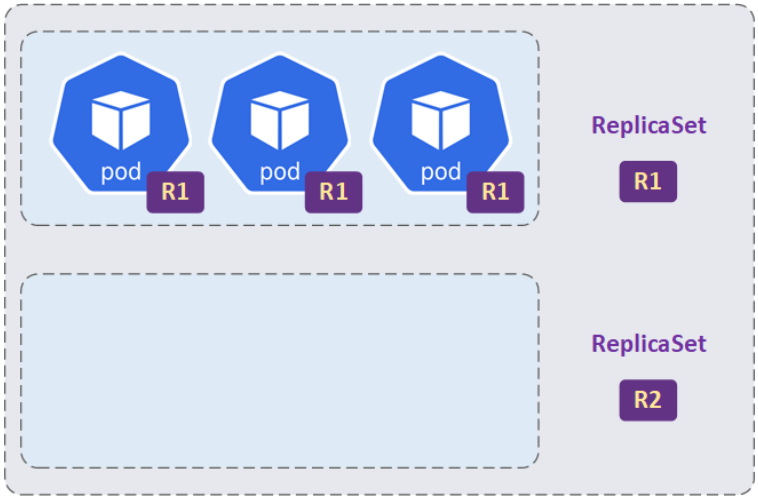
- *key*: Name des Labels, für das die Bedingung gilt.
- *operator*: Der Operator, der die Bedingung definiert. Die gültigen Operatoren sind: *In*, *NotIn*, *Exists* und *DoesNotExist*
- *values:* Ist eine Liste von Werten, die mit dem Label *(key)* verglichen werden. Dies ist nur für die Operatoren In und NotIn relevant
    - *In* - Dieser Operator überprüft, ob der gegebene Key in der Liste der angegebenen Werte enthalten ist.
    - *NotIn* - Dieser Operator ist das Gegenteil von *In* und überprüft, ob der gegebene Key nicht in der Liste der Werte enthalten ist.
    - *Exists* - überprüft, ob ein bestimmtes Label unabhängig von seinem Wert vorhanden ist.
    - *DoesNotExist* - Dies ist das Gegenteil von *Exists*

Deployment

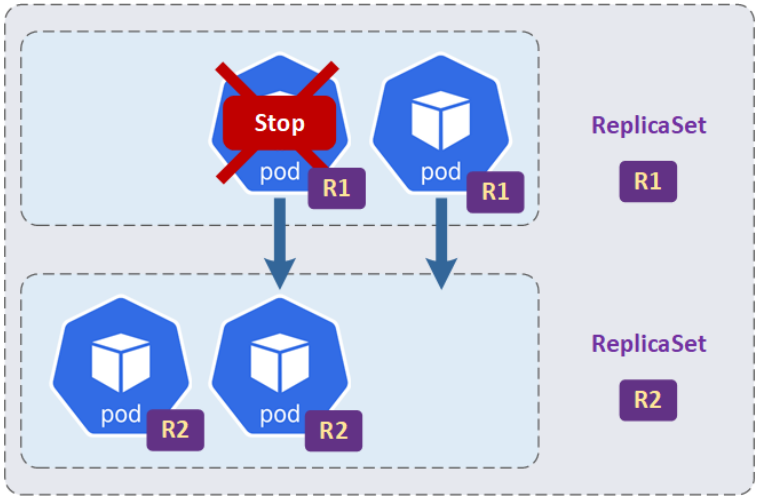
Wie bereits erwähnt, basieren Deployments auf ReplicaSets. Deployments ermöglichen die Aktualisierung oder Änderung von Anwendungen durch die Verwaltung der zugrunde liegenden ReplicaSets. Dies geschieht durch die Erstellung neuer ReplicaSets und die Anpassung ihrer Größe bei gleichzeitiger Verkleinerung bestehender ReplicaSets. Dieser Prozess ermöglicht sogenannte "Rolling Updates". Unter "Rolling Updates" versteht man das schrittweise Hinzufügen neuer Objekte und das Entfernen alter Objekte ohne Ausfallzeiten. Eine weitere wichtige Eigenschaft des Deployments ist die Möglichkeit, die vorgenommenen Änderungen durch ein Rollback wieder rückgängig zu machen.

Die Aktualisierung bzw. das "Rolling Update" von Container-basierten Anwendungen funktioniert wie folgt:

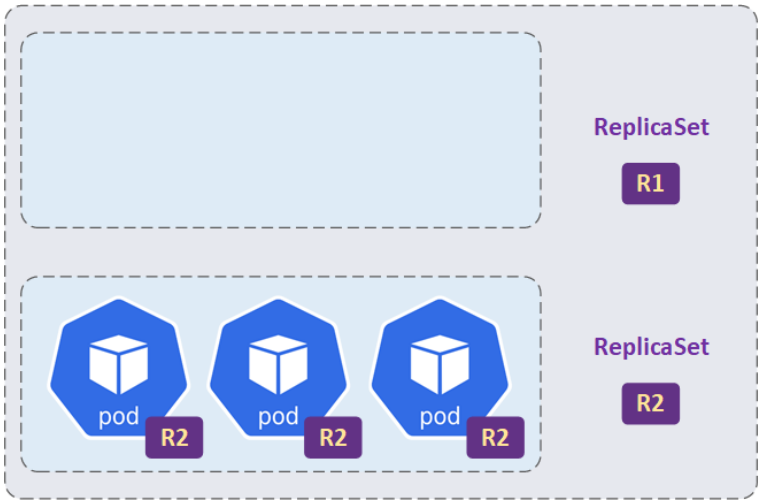
- Im ersten Schritt wird ein neues ReplicaSet für die neue Version der Applikation (neues Container-Image) bereitgestellt. Dieses neue ReplicaSet (R2) enthält zunächst keine Pods.



- Danach startet das Deployment neue Pods im neuen ReplicaSet und stoppt gleichzeitig die alten Pods im alten ReplicaSet. Dies geschieht schrittweise, um sicherzustellen, dass die Anwendung weiterhin verfügbar ist und die Lastverteilung sichergestellt ist.



- Dieser Prozess wird fortgesetzt, bis alle alten Pods gestoppt und durch neue Pods ersetzt wurden. Am Ende dieses Prozesses wird das alte ReplicaSet nicht gelöscht und bleibt mit 0 Pods bestehen. Dies ist notwendig, falls während der Aktualisierung Probleme auftreten und der Rollback-Prozess durchgeführt werden soll.



Deployment verwendet auch dieselben Labels, Selektoren und Operatoren wie ReplicaSets.

ReplicaSet vs. Deployment	
ReplikaSet	**Deployment**
ReplicaSets sind dafür verantwortlich, eine bestimmte Anzahl von Pods einer bestimmten Spezifikation bereitzustellen und aufrechtzuerhalten.	Deployments ermöglicht die Bereitstellung neuer Versionen der Anwendungen durch die schrittweise Erstellung der Pods in den neuen ReplicaSets und die schrittweise Löschung der alten.

RollingUpdate – Parameter

Das RollingUpdate kann durch die beiden Parameter *maxUnavailable* und *maxSurge* gesteuert werden. Der Parameter *maxUnavailable* gibt an, wie viele alte Pods gleichzeitig entfernt werden können. Der Parameter *maxSurge* gibt an, wie viele neue Pods gleichzeitig erzeugt werden können. Beide Parameter sind optional und haben den Standardwert 1.

kubectl Befehle für das Deployment

Hier sind die gängige kubectl Befehle, die dabei helfen, die Deployments im Cluster zu verwalten und zu überwachen.

- *kubectl get deployments* - zeigt eine Liste aller Deployments im Cluster an.
- *kubectl describe deployment <deployment-name>* - gibt eine detaillierte Beschreibung des angegebenen Deployments zurück.
- *kubectl scale deployment <deployment-name> --replicas=<number>* - skaliert die Anzahl der Replikate im Deployment auf die angegebene Anzahl.
- *kubectl rollout status deployment <deployment-name>* - überprüft den Rollout-Status des Deployments.

- kubectl rollout undo deployment <deployment-name> - setzt den Rollout auf eine vorherige Version des Deployments zurück.
- kubectl rollout history deployment <deployment-name> - zeigt eine Liste der Rollout-Historie des Deployments an.
- kubectl delete deployment <deployment-name> - löscht das angegebene Deployment aus dem Cluster.

DaemonSet

Das DaemonSet sorgt dafür, dass eine Kopie eines bestimmten Pods auf allen oder einigen Knoten eines Clusters läuft. Wenn ein neuer Knoten zum Cluster hinzugefügt wird, startet das DaemonSet einen Pod auch auf diesem Knoten. Mit „einigen Knoten“ ist gemeint, dass DaemonSet so konfiguriert werden können, dass sie nur auf bestimmten Knoten laufen.

Die DaemonSets in Kubernetes werden häufig für hauptsächlich technische Zwecke auf den Knoten verwendet. Dazu gehören: Logging-Dienste, Überwachungssysteme, Netzwerkdienste oder Sicherheitsdienste.

Im Gegensatz zu den anderen Kubernetes Controllern (ReplicaSets und Deployments) verwendet das DaemonSet nicht nur *Labels* und *Selectors*, sondern auch das *nodeSelector* oder *nodeAffinity* Attribut, um Pods einem bestimmten Node zuzuordnen.

Beispiel eines YAML-Manifestes für DaemonSet:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: prometheus-daemonset
  namespace: default
spec:
  selector:
    matchLabels:
      name: prometheus
  template:
    metadata:
      labels:
        name: prometheus
    spec:
      nodeSelector:
        disk: ssd
      containers:
      - name: prometheus
        image: prometheus:2.44.0
        ports:
        - containerPort: 80
```

StatefulSet

Ein Kubernetes StatefulSet ist eine weitere Kubernetes-Ressource, die für die Bereitstellung und Skalierung einer Gruppe von Pods verwendet werden kann. Wie der Name StatefulSet schon andeutet, handelt es sich bei StatefulSets um zustandsbehaftete Anwendungen. Die von StatefulSet erzeugten Pod-Instanzen haben eine eindeutige und persistente Identität.

Diese Identität muss auch dann erhalten bleiben, wenn die Pods neu geplant, aktualisiert, gelöscht oder neu erstellt werden. Typische Anwendungsbeispiele für Stateful Pods sind Datenbanken.

Hier sind einige der Hauptmerkmale von Stateful Pods:

- Alle Pods in einem StatefulSet müssen eindeutige und unveränderliche Netzwerkidentifikation (Netzwerknamen) haben.
- Jeder Pod kann einem oder mehreren Persistent Volumes zugewiesen werden. Es muss sichergestellt sein, dass die Zuordnung zu bestimmten Volumes auch nach einem Neustart weiterhin unverändert bleibt.
- Für viele Anwendungen ist es äußerst wichtig, dass Pods in einer strikten Reihenfolge erstellt, skaliert, gelöscht und geschlossen werden. Nur so kann die Konsistenz der Daten gewährleistet werden.

Headless Service

Headless Services werden zusammen mit StatefulSets verwendet. Stateful-Anwendungen erfordern oft eine direkte Kommunikation zwischen den Pods oder von außerhalb des Clusters zu einem bestimmten Pod. Dies wäre mit Mechanismen wie Load Balancer oder ClusterIP kaum umsetzbar, daher wird Cluster DNS verwendet, um die gegenseitige Kommunikation der Pods über den Namen zu ermöglichen.

Hier ist ein einfaches Beispiel für ein Kubernetes StatefulSet mit einem zugehörigen Headless Service

```
apiVersion: v1
kind: Service
metadata:
  name: mein-headless-service
spec:
  clusterIP: None   # None macht diesen Service zu einem Headless Service
  selector:
    app: meine-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376

---
```

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: mein-statefulset
spec:
  serviceName: "mein-headless-service"
  replicas: 3
  selector:
    matchLabels:
      app: meine-app
  template:
    metadata:
      labels:
        app: meine-app
    spec:
      containers:
      - name: mein-container
        image: meine-image
        ports:
        - containerPort: 9090
```

Jobs

Während alle oben genannten Controller-Typen für den Start und den kontinuierlichen Betrieb der Pods vorgesehen sind, besteht der Hauptzweck von Jobs darin, die einzelnen Tasks in einem Kubernetes-Cluster auszuführen.

Die folgenden drei Szenarien beschreiben die Funktionsweise von Jobs am besten:

- Der Job ist dafür verantwortlich, einen oder mehrere Pods innerhalb des Kubernetes-Clusters zu erstellen.
- Er sorgt auch dafür, dass der Pod oder die Pods ein bestimmtes Programm in einem Container ausführen. Normalerweise sollte dieses Programm bis zu seinem „natürlichen Ende“ laufen. Es sei denn, es wird aufgrund eines Fehlers oder aus einem anderen Grund unterbrochen.
- Danach muss der Kubernetes-Job sicherstellen, dass die angegebene Anzahl von Pods ihre Aufgaben erfolgreich abgeschlossen hat. Wenn dies nicht der Fall ist, startet der Job diese Pods neu, um sicherzustellen, dass die Tasks abgeschlossen sind.

Jobs können parallel oder seriell ausgeführt werden und erzeugen mindestens einen Pod, der die Aufgabe ausführt. Nach Beendigung des Jobs werden die Pods automatisch gelöscht.

Hier ist ein Beispiel einer einfachen Wartungsaufgabe. Im Container *mein-container* werden die Daten aus dem Ordner */var/log* gelöscht, wenn sie älter als 7 Tage sind:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: log-cleaner
spec:
  template:
    spec:
      containers:
      - name: log-cleaner
        image: mein-container
        args:
        - /bin/sh
        - -c
        - find /var/log -type f -mtime +7 -delete
      restartPolicy: OnFailure
```

CronJobs

Ein CronJob wird verwendet, um eine bestimmte Aufgabe in regelmäßigen Abständen automatisch auszuführen. Im Gegensatz zu einem "normalen" Job, der nur einmal ausgeführt wird, kann ein CronJob nach einem bestimmten Zeitplan ausgeführt werden. Dieses Konzept ähnelt dem UNIX- oder Linux-CronJob.

Healthcheck-Objekte

Liveness, Readiness und Startup Probes sind Mechanismen in Kubernetes, um die Gesundheit von Containern in einem Pod zu überwachen und sicherzustellen, dass sie ordnungsgemäß funktionieren.

Liveness Probe

Die Liveness Probe prüft, wie der Name schon sagt, ob ein Container noch läuft. Wenn die Liveness Probe fehlschlägt, wird der Container neu gestartet.

Die Überprüfung des Containerstatus kann wie folgt durchgeführt werden: Es wird ein HTTP-Request an eine bestimmte URL oder einen bestimmten Port des Containers gesendet, solange eine erwartete Antwort zurückgegeben wird, wird der Container als „lebendig“ markiert. Es kann auch ein bestimmter Befehl innerhalb des Containers ausgeführt werden.

Hier ist ein Beispiel für eine Liveness Probe, die alle 10 Sekunden eine HTTP-Anfrage an die URL "/healthcheck" sendet:

```
apiVersion: v1
kind: Pod
metadata:
  name: mein-pod
spec:
  containers:
  - name: mein-container
    image: mein-image
    livenessProbe:
      httpGet:
        path: /healthcheck
        port: 80
      periodSeconds: 10
```

Readiness Probe

Die Readiness Probe prüft, ob der Container im Pod bereit ist, eingehende Netzwerkanfragen zu empfangen. In diesem Fall ist die Readiness Probe der Liveness Probe sehr ähnlich, jedoch mit einem wichtigen Unterschied. Wenn die Readiness Probe eines Containers fehlschlägt, wird Kubernetes keinen Netzwerkverkehr mehr an diesen Container senden, aber der Container wird nicht neu gestartet.

Ein typischer Anwendungsfall für eine Readiness Probe wäre, dass eine Anwendung eine gewisse Zeit benötigt, um zu starten (z.B. eine Datenbankverbindung zu einem Backend-Server herzustellen), bevor sie Anfragen bearbeiten kann.

Hier ist ein Beispiel für eine Readiness Probe, die alle 5 Sekunden eine TCP-Verbindung zum Port 8080 herstellt:

```
apiVersion: v1
kind: Pod
metadata:
  name: mein-pod
spec:
  containers:
  - name: mein-container
    image: mein-image
    readinessProbe:
      tcpSocket:
        port: 8080
      periodSeconds: 10
```

Startup Probe

Die Startup-Probe prüft, ob die Anwendung im Container erfolgreich gestartet wurde. Dieser Test wird verwendet, wenn die Anwendung viel Zeit zum Starten benötigt (eine lange Initialisierungsphase hat).

Wenn die Probe erfolgreich ist, wird davon ausgegangen, dass der Container korrekt gestartet wurde. Wenn die Startup-Probe fehlschlägt, wird der Container neu gestartet.

Die Startup Probe kann verhindern, dass Kubernetes den Container ständig neu startet, weil die Liveness Probe fehlschlägt, weil die Anwendung noch nicht initialisiert wurde.

Hier ist ein Beispiel für eine Startup Probe, die einen HTTP-Request an die URL "/startCheck" sendet und den Container als erfolgreich gestartet markiert, wenn eine erfolgreiche Antwort zurückgegeben wird:

```
apiVersion: v1
kind: Pod
metadata:
  name: mein-pod
spec:
  containers:
  - name: mein-container
    image: mein-image
    startupProbe:
      httpGet:
        path: / startCheck
        port: 8080
      failureThreshold: 30
      periodSeconds: 10
```

Taints und Tolerations

Taints und Tolerations sind Mechanismen, die sicherstellen, dass Pods nicht auf ungeeigneten Kubernetes-Knoten geplant oder platziert werden. In Bezug auf Kubernetes kann der Begriff Taints mit *markiert* oder gekennzeichnet übersetzt werden. Taints und Tolerations sind untrennbar miteinander verbunden, da sie als Paar arbeiten. Taints werden zu Knoten hinzugefügt, während Tolerations in der Pod-Spezifikation definiert werden. Wenn ein Taint zu einem Knoten hinzugefügt wird, werden alle Pods abgelehnt, die keine Toleration für diesen Taint haben.

Taints und Tolerations- technische Umsetzung

Die technische Umsetzung ist relativ einfach. Die Worker-Knoten müssen mit entsprechenden Taints versehen werden, z.B. wir wollen erreichen, dass auf den Knoten *wn-01* und *wn-02* nur die Pods ausgeführt werden, die mit dem Taint „*monitoring*“ versehen sind.

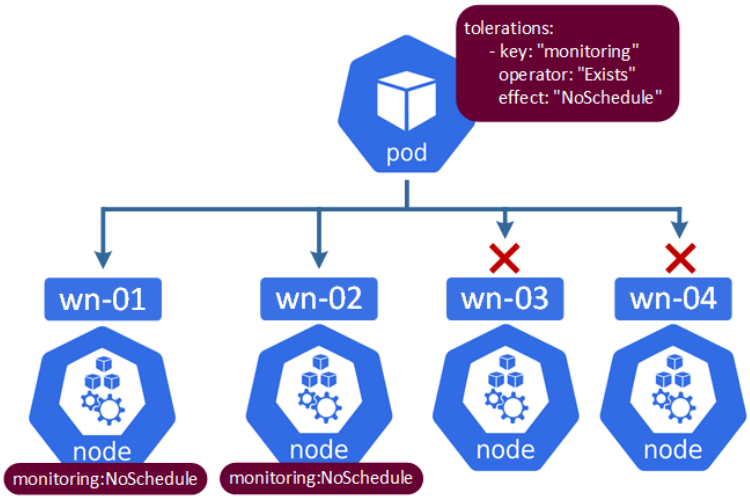
Die untere Befehle werden den Taint "monitoring" zu den Knoten wn-01 und wn-02 hinzufügt. Dabei wird die Option "NoSchedule" verwendet. (wird weiter unter erklärt).

```
kubectl taint nodes wn-01 monitoring:NoSchedule
kubectl taint nodes wn-02 monitoring:NoSchedule
```

Weiterhin muss eine YAML-Datei erstellt werden, die diesen Taint macht.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mein-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mein-app
  template:
    metadata:
      labels:
        app: mein-app
    spec:
      containers:
      - name: mein-app
        image: nginx
      tolerations:
      - key: "monitoring"
        operator: "Exists"
        effect: "NoSchedule"
```

Hier ist die visuelle Darstellung der Konfiguration:



Taint-Optionen

- **NoSchedule** - bedeutet, dass der Kubernetes Scheduler keine neuen Pods auf dem Knoten zulässt, wenn diese den Taint nicht tolerieren. NoSchedule hat keinen Einfluss auf Pods, die bereits auf dem Knoten laufen.
- **PreferNoSchedule** - in diesem Fall versucht der Kubernetes Scheduler die Planung von Pods zu vermeiden, die keine Toleranz für fehlerhafte Knoten haben. Diese Option wird verwendet, wenn keine besser geeigneten Knoten verfügbar sind.

- **NoExecute** – dies ist die härtere Version von „NoSchedule“. NoExecute entfernt (evakuiert) Pods, die bereits auf dem Knoten laufen, wenn sie den Taint nicht tolerieren.

Typische Anwendungsfälle

Der **NoSchedule**-Effekt wird häufig verwendet, wenn die Notwendigkeit besteht, Worker-Knoten für bestimmte Aufgaben oder Benutzergruppen zu reservieren oder sicherzustellen, dass nur bestimmte Pods auf Knoten mit spezieller Hardware (wie z.B. wie GPUs) geplant werden.

Ein Beispiel ist die garantierte Nutzung durch eine bestimmte Benutzergruppe oder die Verwendung einer bestimmten Hardware:

```
kubectl taint nodes <nodename> cityCologne:NoSchedule
kubectl taint nodes <nodename> grafikkarte:NoSchedule
```

Wenn es notwendig und sinnvoll ist, können die Taints ein wenig komplexer sein, z.B:

```
kubectl taint nodes <node-name> grafikkarte=true:NoSchedule
```

In diesem Fall wird die Tolerations in spec so aussehen:

```
spec:
  containers:
  - name: pod-name
    image: image-name
  tolerations:
  - key: "grafikkarte"
    operator: "Equal"
    value: "true"
    effect: "NoSchedule"
```

Der **NoExecute-Effekt** wird in den folgenden Situationen verwendet:

Der andere Anwendungsfall wäre, wenn ein Knoten aufgrund eines Netzwerkausfalls oder aus anderen Gründen nicht erreichbar ist. In diesem Fall kann dem Knoten kein Taint direkt zugewiesen werden, sondern der Zustand des Knotens als wird in der Kubernetes API mit dem Schlüssel "nicht erreichbar" (**node.kubernetes.io/unreachable**) oder "nicht bereit" (**node.kubernetes.io/not-ready**) markiert und dem Effekt *NoExecute* zugewiesen. Die Unterschiede zwischen **unreachable** und **not-ready** werden weiter unten erklärt.

Zusätzlich kann dem Taint eine **tolerationSeconds**-Option hinzugefügt werden. Die Option tolerationSeconds bestimmt, wie lange der Knoten von der Verteilung ausgeschlossen bleibt. Der Standardwert ist 300 Sekunden, kann aber in der Pod-Spezifikation überschrieben werden.

Hier ist die Liste der Labels, die zur Beschreibung des Knoten-Zustandes eingesetzt werden. Die Labels werden nicht nur in Verbindung mit Taints verwendet.

Zustand	Beschreibung
node.kubernetes.io/not-ready	not-ready wird automatisch hinzugefügt, wenn der Status des Kubelet Heartbeats von Ready zu NotReady wechselt. Der mögliche Anwendungsfall könnte z.B. sein, dass der Control-Node den Worker zwar netzwerktechnisch erreicht, aber den Workload nicht ausführen kann.
node.kubernetes.io/unreachable	unreachable wird ebenfall automatisch hinzugefügt, wenn der Kubernetes Control-Nodes keine Verbindung zum Worker-Nodes herstellen kann (Netzwerkprobleme, Worker heruntergefahren, Kubelet läuft nicht)
node.kubernetes.io/unschedulable	unschedulable bedeutet, dass keine neuen Pods auf dem Knoten geplant werden können. Dies kann durch den Administrator oder aufgrund von Ressourcenknappheit festgelegt werden (die drei unteren Punkte).
node.kubernetes.io/network-unavailable	network-unavailable bedeutet, dass das Netzwerk des Knoten nicht verfügbar ist
node.kubernetes.io/memory-pressure	memory-pressure bedeutet, dass der Speicher auf dem Knoten knapp wird und der Workload möglicherweise nicht mehr ausgeführt werden kann.
node.kubernetes.io/disk-pressure	disk-pressure weist auf eine hohe Auslastung der Festplatten auf den Knoten hin
node.kubernetes.io/pid-pressure	pid-pressure bedeutet, dass die Anzahl der noch verfügbaren Prozess-IDs (PIDs) auf dem Knoten knapp wird und neue Prozesse nicht gestartet werden können.

Alle oberen Labels werden abhängig vom Knotenstatus mit dem Befehl *kubectl describe node* angezeigt:

- **not-ready** und **unreachable** im Abschnitt *Taints*.
- **network-unavailable**, **memory-pressure**, **disk-pressure** und **pid-pressure** im Abschnitt Conditions.
- **Unschedulable** kommt separat.

NodeSelector

NodeSelector ist eine relativ einfache Methode, um einen Pod einem bestimmten Node zuzuweisen. Die Konfiguration des NodeSelectors erfolgt durch Zuweisung von Schlüssel-Wert-Paaren in der Pod-Spezifikation.

In unserem Beispiel wollen wir sicherstellen, dass bestimmte Knoten nur auf den Pods laufen, die mit SSDs ausgestattet sind. Zuerst weisen wir einem Node ein Label zu:

```
kubectl label nodes <podname> disktype=ssd
```

So kann die dazugehörige Pod-Konfiguration aussehen:

```
apiVersion: v1
kind: Pod
metadata:
  name: mein-pod
spec:
  containers:
  - name: mein-container
    image: mein-nginx
  nodeSelector:
    disktype: ssd
```

Node Affinity

Node Affinity kann als Erweiterung von NodeSelector angesehen werden. Node Affinity bietet mehr Flexibilität und kann zwischen „erforderlichen" oder „harten“ Regeln und "bevorzugten" oder „weichen“ Regeln unterscheiden.

Die Trennung zwischen den erforderlichen und den bevorzugten Anforderungen erfolgt auf der Grundlage dieser Optionen:

requiredDuringSchedulingIgnoredDuringExecution - harte Anforderung

preferredDuringSchedulingIgnoredDuringExecution - weiche Anforderung

Die folgenden Beispiele veranschaulichen diese Optionen:

Labels hinzufügen:

```
kubectl label nodes <node-name> disk=ssd-sata6
kubectl label nodes <node-name> disk=ssd-nvme
```

Aufgrund der harten Anforderung können Pods nur auf Knoten mit dem Label „*disk*“ und dem Wert "*ssd-nvme*" geplant werden. Wenn kein solcher Knoten vorhanden ist, werden die Pods nirgendwo gestartet.

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: prometheus-daemonset
  namespace: default
spec:
  selector:
    matchLabels:
      name: prometheus
  template:
    metadata:
      labels:
        name: prometheus
    spec:
```

```
    affinity:
      nodeAffinity:
        requiredDuringSchedulingIgnoredDuringExecution:
          nodeSelectorTerms:
          - matchExpressions:
            - key: disk
              operator: In
              values:
              - ssd-nvme
    containers:
    - name: prometheus
      image: prometheus:2.44.0
      ports:
      - containerPort: 80
```

Aufgrund der weichen Anforderung werden Pods auf Knoten mit dem Label „*disk*“ und dem Wert "*ssd-sata6*" eingeplant, wenn diese vorhanden sind. Ist dies nicht der Fall, werden beliebige Knoten verwendet.

```
kind: DaemonSet
metadata:
  name: prometheus-daemonset
  namespace: default
spec:
  selector:
    matchLabels:
      name: prometheus
  template:
    metadata:
      labels:
        name: prometheus
    spec:
      affinity:
        nodeAffinity:
          preferredDuringSchedulingIgnoredDuringExecution:
          - weight: 1
            preference:
              matchExpressions:
              - key: disk
                operator: In
                values:
                - ssd-sata6
      containers:
      - name: prometheus
        image: prometheus:2.44.0
        ports:
        - containerPort: 80
```

Der Parameter **weight** wird in **preferredDuringSchedulingIgnoredDuringExecution** verwendet und kann einen Wert zwischen 1 und 100 haben. Der Wert stellt die Priorität dar, je höher der Wert, desto mehr wird dieses Pod vom Scheduler bevorzugt.

Damit der Parameter **weight** richtig verwendet werden kann, sollten die Optionen (preferredDuringSchedulingIgnoredDuringExecution) mit mehreren Auswahlmöglichkeiten versehen und entsprechend gewichtet werden. So sieht eine mögliche Konfiguration aus:

```
preferredDuringSchedulingIgnoredDuringExecution:
- weight: 1
  preference:
    matchExpressions:
    - key: disk
      operator: In
      values:
      - ssd-sata6
- weight: 5
  preference:
    matchExpressions:
    - key: disk
      operator: In
      values:
      - ssd-nvme
```

Pod Affinity / Pod Anti-Affinity

Pod Affinity

Pod Affinity stellt sicher, dass einzelne Pods oder eine Gruppe von Pods auf demselben Knoten (oder einer Gruppe von Knoten) ausgeführt werden. Dies ist für Anwendungen interessant, die in einer gewissen Abhängigkeit zueinander stehen und eine optimale Netzwerkkommunikation benötigen.

Analog zu den Node Affinity verwenden Pod Affinity gleiche Mechanismen:

- requiredDuringSchedulingIgnoredDuringExecution
- preferredDuringSchedulingIgnoredDuringExecution

Im folgenden Beispiel wird der Scheduler versuchen, die drei Pods des Webservers auf dem gleichen Knoten zu platzieren, auf dem bereits die Datenbank ("app=database") läuft.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
spec:
  replicas: 3
  selector:
```

```
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      affinity:
        podAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - database
            topologyKey: "kubernetes.io/hostname"
      containers:
      - name: web-server
        image: web-server-container
```

Die Bindung an einen Knoten wird durch das Schlüssel-Wert-Paar bestimmt: *topologyKey: "kubernetes.io/hostname"*. Die Manifestdatei für das Datenbank-Deployment muss dementsprechend das Schlüssel-Wert-Paar **"app: database"** enthalten. Mögliche Operatoren sind: *In, NotIn, Exists, DoesNotExist, Lt, Gt.*

Datenbank-Deployment:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: database
spec:
  replicas: 1
  selector:
    matchLabels:
      app: database
  template:
    metadata:
      labels:
        app: database
    spec:
      containers:
      - name: database
        image: meine-database
```

Hier ist ein weiteres Beispiel von der offiziellen Kubernetes-Seite. In diesem Fall erfolgt die Platzierung der Pods anhand eines Topologie-Labels "topology.kubernetes.io/zone" (geografische Zone eines Cloud-Providers). Pods werden entweder in europa-nord oder europa-west platziert.

```
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: topology.kubernetes.io/zone
          operator: In
          values:
          - europa-nord
          - europa-west
```

Pod Anti-Affinity

Pod Anti-Affinity ist natürlich das Gegenteil von Pod Affinity. Mit Pod Anti-Affinity kann sichergestellt werden, dass bestimmte Pods nicht auf demselben Knoten laufen. Diese Option kann bei der Planung von Hochverfügbarkeit nützlich sein. Mit Pod Anti-Affinity kann verhindert werden, dass Pods, die den gleichen Dienst ausführen, auf dem gleichen Node laufen.

Auch hier gelten dieselben Mechanismen:

- requiredDuringSchedulingIgnoredDuringExecution
- preferredDuringSchedulingIgnoredDuringExecution

Kubernetes Netzwerk

Das Kubernetes-Netzwerk ist ein virtuelles Netzwerk, welches in erster Linie dazu dient, die Kommunikation zwischen den Pods und anderen Services/Komponenten im Kubernetes Cluster zu ermöglichen.

Das Kubernetes-Netzwerk unterscheidet sich von anderen Netzwerktypen in mehreren Bereichen:

CNI-Plugin

Das CNI-Plugin (CNI steht für Container Networking Interface) ist in Grunde genommen eine Spezifikation für verschiedene Netzwerk-Plugins in Kubernetes. Das CNI-Plugin selbst dient zur Verwaltung der Netzwerkverbindungen zwischen Pods und Diensten innerhalb des Clusters. Es existiert bereits eine breite Palette von CNI-Plugins, die die Basis-Funktionalität wesentlich erweitern können. Mehr dazu auf dieser Seite www.github.com/cni

Service-Discovery

Service-Discovery wie der Name auch sagt, dient zur Erkennung der Services innerhalb eines Clusters, besonders wenn diese aus mehreren Containern bestehen und auch auf verschiedenen Knoten laufen. Außerdem ermöglicht Service-Discovery den Pods, andere Pods und Services im Cluster zu finden, ohne spezifische IP-Adressen kennen zu müssen.

Network-Policies

Die Network-Policies können den Netzwerkverkehr zwischen den Pods und Services im Kubernetes Cluster steuern. Die ermöglichen auch den Netzwerk-Traffic basierend auf IP-Adressen, Ports und anderen Merkmalen zu filtern.

Pod-to-Pod-Kommunikation

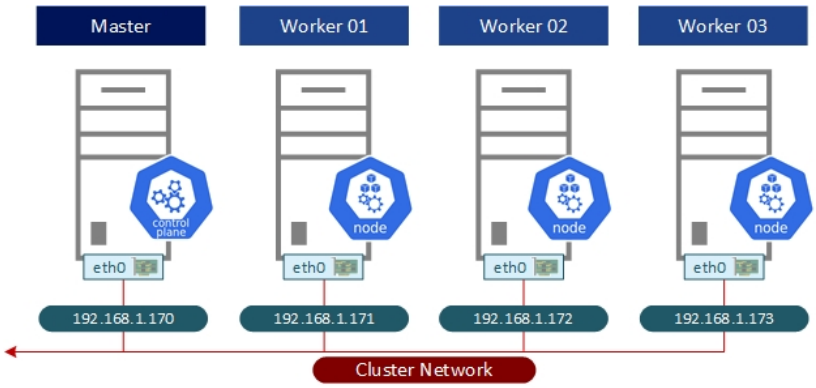
Die Pods, die auf verschiedenen Nodes aufgeführt werden, können miteinander ohne zusätzlichen speziellen Konfigurationen kommunizieren.

Netzwerk-Arten im Kubernetes Cluster

In **einem** Kubernetes-Cluster gibt es drei Arten von Netzwerken: das **Node Network**, **das Pod Network / Cluster Network** und das **Service Network**. Jedes Netzwerk erfüllt seine spezifische Rolle im Cluster.

- Node Network - Kommunikation zwischen den verschiedenen Worker- und Control-Knoten im Cluster.
- Pod Network - Kommunikation zwischen den einzelnen Pods im Cluster.
- Service Network - Abstrakte Schicht, die stabile Netzwerkzugriffspunkte für einen oder mehrere Pods bereitstellt.

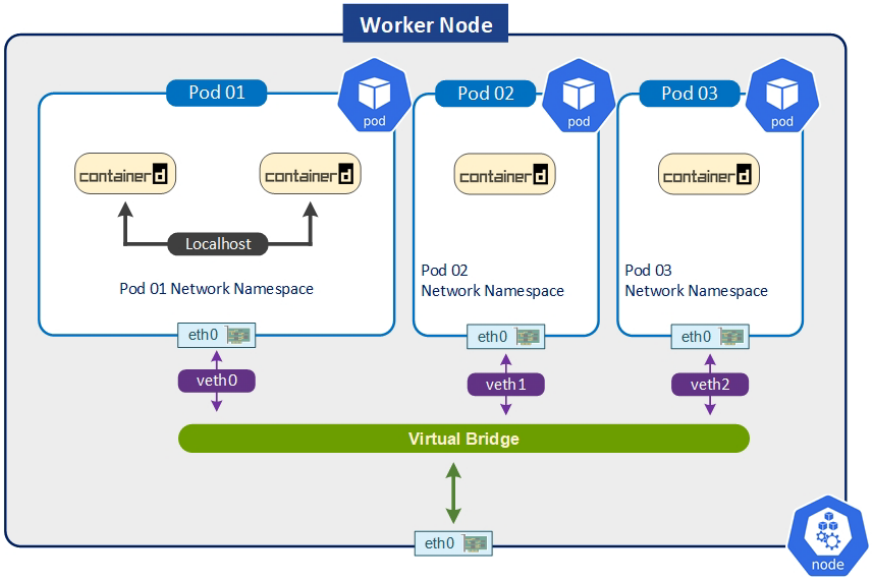
Node Network



Das Node Network wird von Kubernetes verwendet, um die Kommunikation zwischen den Nodes im Cluster zu ermöglichen. Das Node Network wird für folgende Zwecke genutzt: den Clusterzustand zu synchronisieren, zur Lastverteilung zwischen den Nodes, für den Zugriff auf gemeinsam genutzte Ressourcen wie z.B. Storage.

```
anatoli@admin-vm:~/cert-manager$ kubectl get nodes -o wide
NAME   STATUS   ROLES           AGE   VERSION   INTERNAL-IP     EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION      CONTAINER-RUNTIME
cn01   Ready    control-plane   18d   v1.25.5   192.168.1.170   <none>        Ubuntu 20.04.5 LTS   5.4.0-137-generic   containerd://1.6.12
wn01   Ready    <none>          18d   v1.25.5   192.168.1.171   <none>        Ubuntu 20.04.5 LTS   5.4.0-137-generic   containerd://1.6.12
wn02   Ready    <none>          17d   v1.25.5   192.168.1.172   <none>        Ubuntu 20.04.5 LTS   5.4.0-137-generic   containerd://1.6.12
wn03   Ready    <none>          17d   v1.25.5   192.168.1.173   <none>        Ubuntu 20.04.5 LTS   5.4.0-137-generic   containerd://1.6.12
```

Pod Network / Cluster Network



Das Pod Network dient verständlicherweise für die Kommunikation zwischen den Pods in einem Cluster, unabhängig davon auf welchen Cluster diese ausgeführt werden. Jeder Pod im Cluster hat seine eindeutige IP-Adresse, die innerhalb des gesamten Clusters nur einmal vorkommt. Das Pod-Netzwerk in Kubernetes wird durch CNI-Plugins bereitgestellt und verwaltet. Die Funktionsweise von CNI-Plugins wird weiter näher erläutert.

Die Begriffe "Cluster-Network" und "Pod-Network" werden oft als Synonyme gesehen. Da es in beiden Fällen um die Bereiche handelt, die für die Kommunikation zwischen Pods im Cluster verwendet werden.

Auf dem unteren Bild werden die IP-Adressen des Pod-Netzwerks angezeigt, die bei der Konfiguration von Calico-PlugIn eingegeben wurden.

```
anatoli@admin-vm:~$ kubectl get pods -o wide
NAME                          READY   STATUS    RESTARTS   AGE   IP               NODE   NOMINATED NODE   READINESS GATES
game-2048-64f48f4cd-dtcnj     1/1     Running   0          7d    172.16.198.131   wn03   <none>           <none>
game-2048-64f48f4cd-fxnsq     1/1     Running   0          7d    172.16.189.3     wn02   <none>           <none>
hello-world-7d7874c757-44png  1/1     Running   0          90s   172.16.198.132   wn03   <none>           <none>
hello-world-7d7874c757-mzt2t  1/1     Running   0          90s   172.16.53.195    wn01   <none>           <none>
hello-world-7d7874c757-qtf6t  1/1     Running   0          90s   172.16.189.5     wn02   <none>           <none>
```

Service Network

Dieser IP-Bereich wird für die Kubernetes-Services verwendet. Der Zweck des Service-Netzwerks besteht darin, einem oder mehreren Pods, die den gleichen Service repräsentieren, eine eindeutige IP-Adresse und einen eindeutigen DNS-Namen zuzuweisen.

Wenn die Konfiguration von Service Network nicht geändert wurde, werden die IP-Adresse aus dem Bereich 10.96.0.0/12 zugewiesen. So lässt sich die Konfiguration verifizieren:

kubectl get pods -n kube-system
kubectl describe pod <API_SERVER_POD_NAME> -n kube-system
Nach „--service-cluster-ip-range “suchen.

Service Networks beinhaltet eine Reihe von Komponenten oder bessergesagt dazugehörigen Objekten:

Service-Objekt ist sozusagen ein Kern-Objekt des Service Networks und sorgt dafür, dass die zugrunde liegenden Pods über dauerhaften IP-Adressen oder DNS-Namen (abhängig vom Service-Typ) erreichbar sind. Dadurch wird erreicht, dass die IP-Adresse oder der DNS-Name des Services unverändert bleibt, auch dann, wenn die IP-Adressen der zugrundeliegenden Pods geändert werden.

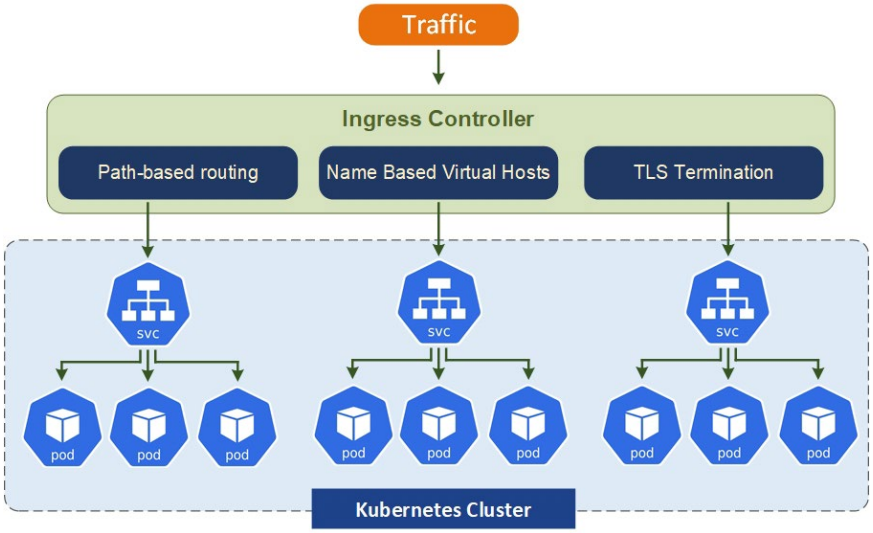
Label-Selectors sorgen dafür, dass z.B. der Datenverkehr zu einer Gruppe von Pods geleitet wird, wenn diese Pods mit entsprechenden Labels versehen sind.

Load Balancing diese Option ist eigentlich selbsterklärend. LB sorgt für den gleichmäßigen Lastausgleich zwischen den verfügbaren Pods, die zum selben Service gehören.

Service-Typs es handelt sich dabei um die Methode, wie auf einen Service (z.B. ClusterIP, NodePort usw.) zugegriffen wird. Mehr dazu in Teil 2 des Beitrages.

EndpointSlice ist ein API-Objekt, welches von Control Nodes erstellt wird und Informationen über verfügbare Dienstinstanzen (Pods) und deren IP-Adressen/Ports speichert. Die Funktionsweise von EndpointSlice ist besonders dann sichtbar, wenn ein Service aus mehreren Pods besteht. EndpointSlice hilft dabei, den eingehenden Datenverkehr auf die Pods zu verteilen, indem es die Zuordnung von IP-Adressen und Ports zu den Pods organisiert und bereitstellt.

Kubernetes Ingress



Ingress ist eine Kubernetes-Ressource, die den Zugriff auf Services eines Clusters von außen ermöglicht. Ingress bietet mehrere Optionen zur Steuerung des eingehenden Netzwerkverkehrs. Der Traffic kann basierend auf Namen, Pfaden oder verschiedene Dienste innerhalb des Clusters verteilt werden. Ingress kann die Aufgaben eines Reverse Proxy übernehmen und somit den Zugriff auf einen oder mehrere interne Server von außen steuern. Außerdem kann zur SSL-Terminierung verwendet werden, um den SSL-Datenverkehr an einen Service weiterzuleiten.

Des Weiteren werden einige Ingress-Funktionsweisen erklärt:

Name-based Virtual Hosts

Ingress kann den eingehenden Datenverkehr basierend auf Hostnamen auf unterschiedliche Services innerhalb desselben Clusters verteilen. Diese Methode ist nützlich, wenn mehrere Anwendungen auf derselben Infrastruktur ausgeführt werden, z.B. wenn eine Anwendung unter *app1.demo.de* und eine andere unter *app2.demo.de* verfügbar sein sollte.

Path-based Routing

Ingress kann auch das Routing des eingehenden Datenverkehrs basierend auf bestimmten Pfaden bedienen. Das bedeutet, dass unterschiedliche Pfade auf verschiedene Services innerhalb des Clusters verweisen können, die aber auf denselben Cluster-IP-Adressen laufen. So lässt sich folgendes Szenario umsetzen, dass eine Anfrage an */app1* auf einen Service und eine Anfrage an */app2* auf einen anderen Service geroutet wird.

TLS Termination

Wie oben bereits erwähnt, kann auch Ingress als TLS-Terminierungspunkt dienen. Ingress kann die Entschlüsselung des eingehenden Datenverkehrs übernehmen und die Daten an den entsprechenden Service weiterleiten.

Service API vs. Ingress

Der Hauptunterschied zwischen der Kubernetes Service API und Ingress besteht darin, dass Kubernetes Service API auf OSI Layer 4 (Transport Layer) und Ingress auf OSI Layer 7 (Application Layer) arbeitet, und somit eine erweiterte Funktionalität für die Verarbeitung von

HTTP-Verkehr bietet. Wenn es bei OSI Layer 4 hauptsächlich darum geht, die Datenpakete erfolgreich vom Sender zum Empfänger zu übertragen, beschäftigt sich Layer 7 mit den Anwendungen, die auf dieser Ebene tätig sind (z.B. E-Mail, Webbrowser usw.). Auch die Implementierung von Load Balancing Algorithmen oder TLS-Verschlüsselung findet auf der Anwendungsebene statt.

Zusätzlich zum Ingress-Controller etabliert sich die Gateway API als moderner Standard für das Traffic-Management, da sie rollenbasierte Konfigurationen und erweiterte Routing-Funktionen bietet.

Open Source Ingress Controller

Es gibt verschiedene Open-Source-Ingress-Controller, die mit Kubernetes verwendet werden können. Die drei verbreitetste sind: Nginx, Traefik und Contour. Wobei Nginx Ingress Controller scheint der populärste von allen zu sein. Alle drei Controller haben ähnliche Funktionen (Reverse-Proxy, Load-Balancer, TLS-Terminierung) und unterscheiden sich in der Architektur und Implementierung. Die Wahl hängt von den spezifischen Anforderungen und Vorlieben ab.

Ingress Beispiel

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: demo.de
    http:
      paths:
      - path: /app
      pathType: Prefix
      backend:
        service:
        name: test-service
        port:
          name: http
```

Im oberen Beispiel wird ein Ingress Controller mit dem Namen *test-ingress* erstellt. Der Hostnamen *demo.de* wird verwendet. Alle Anfragen, die auf den Pfad */app* beginnen, werden an den Service mit namens *test-service* und dem Port *http* weitergeleitet. Die Annotation *nginx.ingress.kubernetes.io/rewrite-target: /* sorgt dafür, dass alle eingehende Anfragen an den Pfad */* der Ziel-Service weitergeleitet werden.

Kubernetes Egress

Egress ist ein Antipode von Ingress. Egress beschäftigt sich verständlicherweise mit den ausgehenden Datenverkehr zu den externen Ressourcen oder APIs. Mit Egress lassen sich ebenfalls unterschiedliche Regeln definieren, um den Datenverkehr auf der Basis

verschiedenen Attributen wie z.B. IP-Adressbereiche, Ports, Protokolle, Namespace-, Pod-, oder Service-Labels zu steuern. Die Egress-Regeln werden in den Network Policies konfiguriert.

Egress Beispiel

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-egress
spec:
  podSelector: {}
  policyTypes:
  - Egress
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
        except:
        - 10.0.0.2/32
```

In diesem Bespiel wird eine Network Policy namens ***allow-egress***. Die Policy wird auf alle Pods angewendet. Der Policy-Typ ***Egress*** zeigt, dass es sich um den ausgehenden Datenverkehr handelt. In diesem Beispiel darf der ausgehende Datenverkehr nur an einen bestimmten IP-Bereich (in diesem Fall 10.0.0.0/24) geleitet werden, außer einer Ausnahme (10.0.0.2).

Kubernetes DNS

Kubernetes DNS (Domain Name System) ist eine integrierte Funktion von Kubernetes, die es Containern und Services im Cluster ermöglicht, über DNS-Namen, anstatt über IP-Adressen zu kommunizieren. Dadurch kann die Kommunikation zwischen Containern und Services vereinfacht werden. (ab der Version 1.21 ist nicht mehr supportet)

Im Kubernetes Cluster wird der DNS-Service in diesem Format "*service.namespace.svc.cluster.local*" verwendet.

Kubernetes CoreDNS

Kubernetes CoreDNS ist ein spezielles DNS-Plugin, das als Standard-DNS-Server in Kubernetes-Clustern verwendet wird. CoreDNS erweitert die Basis-Funktionen und gibt z.B. die Möglichkeit, mehrere Domainnamen zu verwenden und DNS-Anfragen auf externe DNS weiterzuleiten. Ab der Version v1.26 ist CoreDNS die einzige unterstützte Cluster-DNS-Anwendung.

Hier sind die Vorteile aus der offiziellen Seite von CoreDNS: https://coredns.io

- **Flexibilität:** CoreDNS ist sehr flexibel und ermöglicht es Benutzern, benutzerdefinierte DNS-Server zu erstellen.

- **Einfachheit:** CoreDNS verwendet einfache und verständliche textbasierte Konfigurationsdateien Corefile).
- **Performance:** CoreDNS ist darauf ausgelegt eine hohe Leistung und geringe Latenzzeiten zu bieten.
- **Erweiterbarkeit:** Die CoreDNS Plugin-Architektur gibt es die Möglichkeit, neue Funktionen hinzuzufügen oder vorhandene Funktionen zu erweitern.
- **DNS-Protokolle**: CoreDNS unterstützt sowohl DNS-over-HTTP (DoH) als auch DNS-over-TLS (DoT)

So lassen sich DNS- oder CoreDNS-Pods anzeigen:

kubectl get pods --namespace=kube-system -l k8s-app=kube-dns

Kubernetes Netzwerk-Plugins / Container Networking Interface (CNI)

Wenn man über Kubernetes Netzwerk-Plugins und CNI-Plugins spricht, ist meist dasselbe gemeint. CNI ist auch als eine Spezifikation zu verstehen, die von der Cloud Native Computing Foundation (CNCF) entwickelt wurde. Die Plugins erweitern die integrierte Netzwerkfunktionalität von Kubernetes wie z. B. Cluster-IP, NodePort oder LoadBalancer. Besonders in einem großen Cluster (mit vielen Pods und Nodes) kann die Nutzung von Plugins für bessere Skalierbarkeit und effizientere Lastverteilung sorgen sowie im Bereich Sicherheit viele Vorteile bieten. Die CNI-Plugins können verschiedene Arten von Netzwerken bereitstellen: Overlay-Netzwerke, Bridge-Netzwerke und Layer 3-Netzwerke.

Kubernetes unterstützt eine Vielzahl von CNI-Plugins, darunter Flannel, Cilium, Calico, Weave Net und andere. Die Wahl des CNI-Plugins hängt von den spezifischen Anforderungen und der Infrastruktur des Kubernetes-Clusters ab. Weitere Information zum Thema: www.cni.dev und www.github.com/containernetworking/plugins

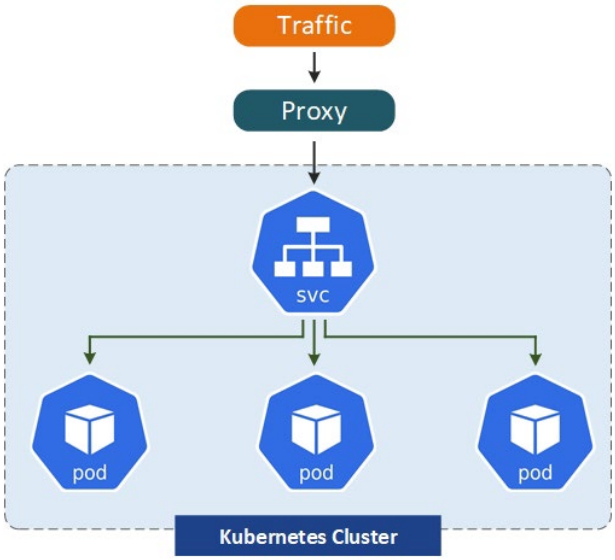
Service-Typen

Wie wir bereits wissen, werden Anwendungen in Kubernetes innerhalb von Pods bereitgestellt und müssen sowohl untereinander als auch nach außen kommunizieren. Um diese Kommunikation zu ermöglichen und zu vereinfachen, stellt Kubernetes verschiedene Service-Typen zur Verfügung. Die Service-Typen dienen als Abstraktionsschicht, um eine Netzwerkverbindung zwischen den verschiedenen Pods innerhalb des Clusters und Objekten außerhalb des Clusters bereitzustellen.

Weiter geht es um folgende Kubernetes-Komponenten bzw. Service-Typen:

- ClusterIP
- NodePort
- LoadBalancer
- ExternalIPs
- ExternalName

ClusterIP

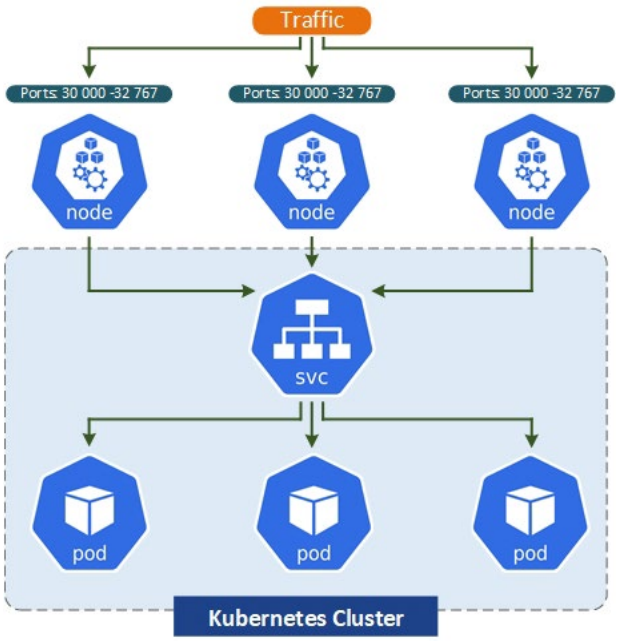


Die Cluster-IP ist eine Art virtuelle IP-Adresse, die einem Kubernetes-Service zugewiesen wird. Diese IP-Adresse ist nur innerhalb des Clusters verfügbar und ermöglicht anderen Pods und Services den Zugriff auf den Service. ClusterIP ist der Standard-Servicetyp in Kubernetes.

Beispiel ClusterIP

```
apiVersion: v1
kind: Service
metadata:
  name: test-clusterip-service
spec:
  selector:
    app: test-app
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

NodePort

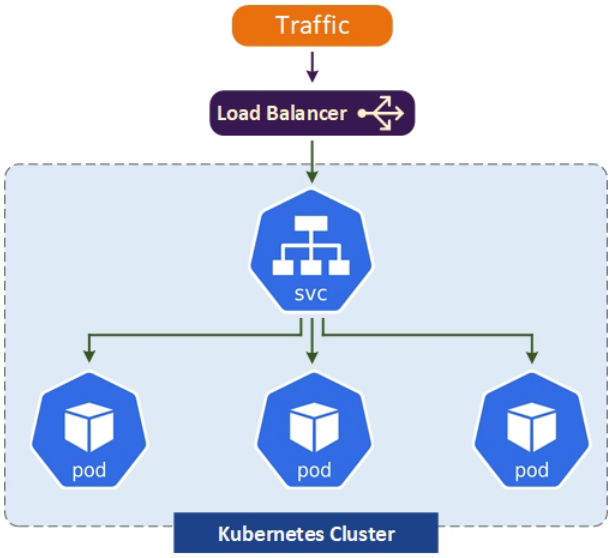


NodePort ist eine Netzwerkfunktion, die es einem Service ermöglicht, über eine Portnummer auf jedem Node im Cluster verfügbar zu sein. Der für den NodePort verwende Port wird automatisch aus einem Bereich zwischen 30000 bis 32767 ausgewählt. Wenn ein Client auf den Service zugreifen möchte, muss er nur die IP-Adresse eines der Nodes im Cluster und den zugewiesenen NodePort verwenden.

Beispiel NodePort

```
apiVersion: v1
kind: Service
metadata:
  name: test-nodeport-service
spec:
  selector:
    app: test-app
  type: NodePort
  ports:
    - name: http
      port: 80
      targetPort: 8080
      nodePort: 30000
```

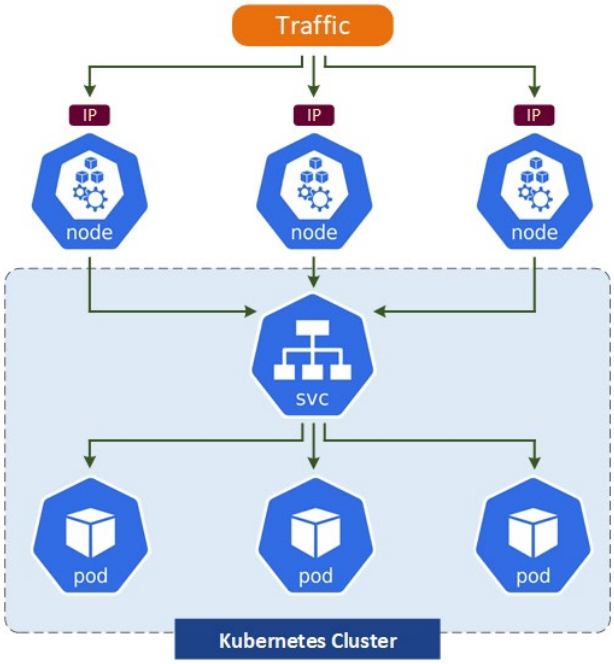
LoadBalancer



Der Service-Typ LoadBalancer bietet einem Service die Möglichkeit, über eine externe Load-Balancer-IP-Adresse erreichbar zu sein. Wenn ein Client auf die Load-Balancer-IP-Adresse zugreift, wird der Datenverkehr an den Service im Cluster weitergeleitet.

LoadBalancer wird normalerweise von einem Cloud-Anbieter bereitgestellt. Welche Load-Balancing-Methoden (z.B. Round Robin, Least Connection, Least Bandwidth, Least Response Time usw.) verwendet werden, hängt von der Implementierung des Load Balancers ab. In vielen Fällen wird Round-Robin-Load-Balancing eingesetzt.

ExternalIPs



Wie Sie auf dem oberen Bild erkennen können, weisen die Service-Typen ExternalIPs und NodePort gewisse Ähnlichkeiten auf. ExternalIPs ermöglicht den Zugriff auf einen internen

Dienst über eine externe IP-Adresse, die verständlicherweise nicht vom Kubernetes-Cluster selbst verwaltet wird. Bei Verwendung von ExternalIPs wird der Datenverkehr von den angegebenen externen IP-Adressen direkt auf die Pods im Cluster weitergeleitet.

Diese Funktion kann nützlich sein, wenn man einen Service für den Zugriff von außen bereitstellen möchte, ohne dabei einen LoadBalancer zu verwenden. Auch wenn bestimmte IP-Adressen für den Zugriff auf den Service zugelassen werden müssen, kann diese Funktion sinnvoll sein.

Beispiel ExternalName

Sie können eine oder mehrere externe IP-Adressen in der Service-Definition angeben, damit der Service über diese Adressen erreichbar ist. Die IP-Adresse 10.10. 20.10 muss außerhalb des Kubernetes-Clusters verwaltet werden.

```
apiVersion: v1
kind: Service
metadata:
  name: test-externalip-service
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
    port: 80
    targetPort: 8080
  externalIPs:
   - 10.10.20.10
```

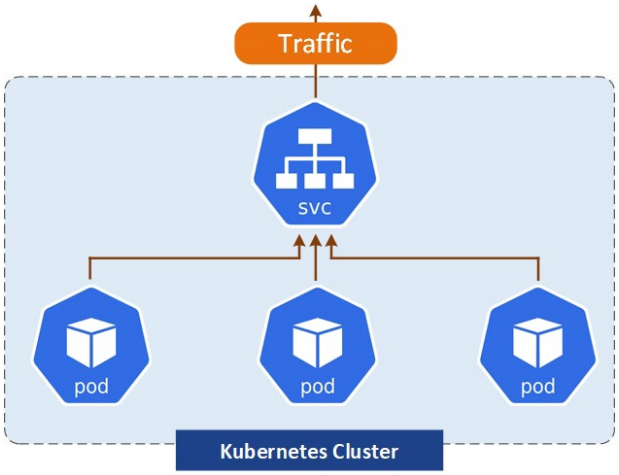
So sieht es aus, wenn drei IP-Adressen (wie auf dem Bild) konfiguriert werden:

```
  externalIPs:
   - 10.10.20.10
   - 10.10.20.11
   - 10.10.20.12
```

NodePort vs. ExternalIPs

Obwohl NodePort und ExternalIPs auf den ersten Blick ähnlich erscheinen, gibt es gravierende Unterschiede zwischen den beiden Service-Typen. Bei NodePort werden die Ports auf jedem Knoten im Cluster automatisch von Kubernetes reserviert und verwaltet. Dabei ist keine manuelle Konfiguration erforderlich, aber die Anzahl der verfügbaren Portbereiche für NodePort-Dienste ist beschränkt. Bei ExternalIPs hingegen müssen eine oder mehrere externe IP-Adressen manuell konfiguriert werden. Die Anzahl der Portbereiche ist aber nicht begrenzt.

ExternalName



Wie man auf dem Bild erkennen kann, leitet der ExternalName den Datenverkehr in eine umgekehrte Richtung, nämlich nach außen. Dies macht den Service-Typ ExternalName zu einer besondere Art von Service in Kubernetes. Der ExternalName kommt zum Einsatz, wenn Anwendungen innerhalb des Clusters auf externe Dienste zugreifen müssen, ohne dass IPs/Ports direkt in den Konfigurationen verwaltet werden.

Bei der Erstellung eines ExternalName werden keine Selektoren (selector) sondern externen DNS-Namen verwendet. Wenn ein Clients (Pod/Container) innerhalb des Clusters diesen Service ansprechen, wird der angegebene DNS-Name anstelle der Service-Adresse zurückgegeben.

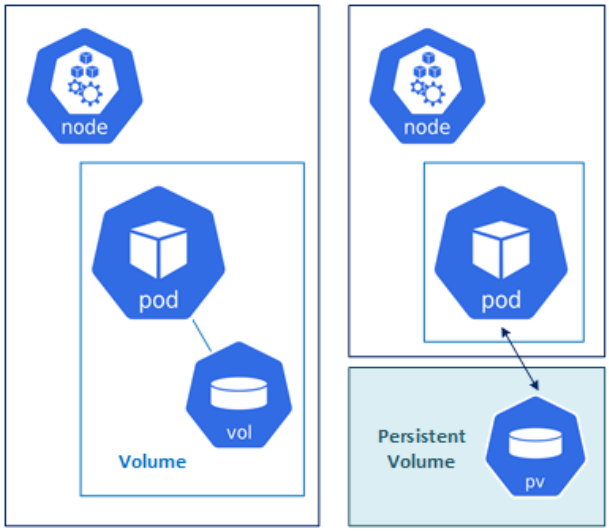
Beispiel ExternalName

In unteren Beispiel erstellt der ExternalName-Service einen DNS-Eintrag für ***test-db-service.svc.cluster.local***. Wenn irgendein Pod innerhalb des Clusters versucht, den Service unter dieser Adresse aufzurufen, wird die Anfrage an ***test-db.demo.lab*** weitergeleitet.

```
apiVersion: v1
kind: Service
metadata:
  name: test-db-service
spec:
  type: ExternalName
  externalName: test-db.demo.lab
  ports:
    - name: db-port
    port: 3306
    protocol: TCP
```

Kubernetes Storage

Zu Beginn ihrer technologischen Entwicklung waren Container ausschließlich für die Bereitstellung zustandsloser Anwendungen vorgesehen. Das bedeutet, dass kurzlebige Objekte nur in der beschreibbaren Schicht eines Containers (Pods) existieren und jedes Mal verloren gehen, wenn ein Pod (in dem ein Container läuft) „zerstört“ und neu erzeugt wird. Daher war es notwendig, eine Technologie zu entwickeln, die sicherstellt, dass die Anwendungsdaten auch nach einem Recreate, oder besser gesagt, während des gesamten Lebenszyklus eines Pods, verfügbar bleiben.



Folgende Objekte helfen uns, die beschriebene Herausforderung zu meistern:

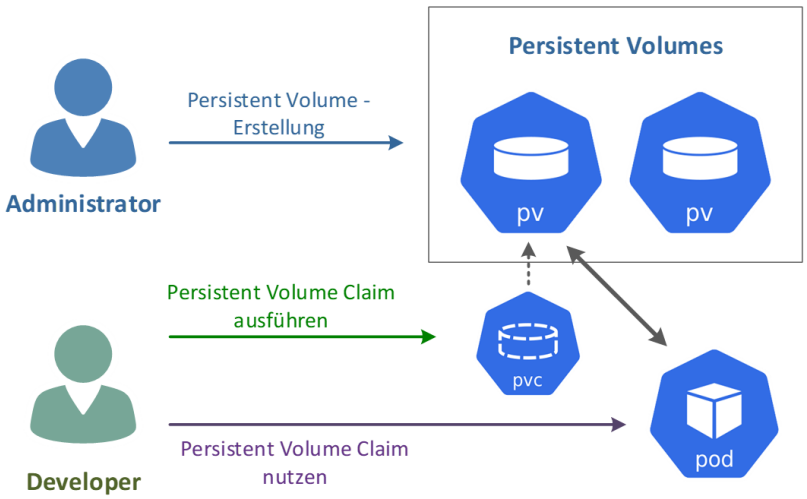
- Volume
- Persistent Volume
- Persistent Volume Claim
- Storage Class
- Access Modes

Im Laufe der Jahre wurden immer mehr stateful (zustandsbehaftet) Applikationen in containerisierter Form bereitgestellt, besonders die Applikationen, die eine Datenbank benötigen.

Volumes

Wie Sie aus der vorherigen Ausführung bereits entnehmen konnten, werden die Volumes nur zum Speichern temporärer Daten während der Lebenszeit eines Pods verwendet. Wenn der Pod gelöscht wird, werden auch alle dazu zugehörige Volumes ebenfalls gelöscht.

Persistent Volume



Das Persistent Volume (PV) ist ein API-Objekt, welches den eigentlichen Speicher darstellt. Wie der Name schon sagt, ist der Lebenszyklus eines Persistent Volumes völlig unabhängig vom Lebenszyklus eines Pods. Ein Persistent Volume kann von einem Administrator oder dynamisch in einem Kubernetes Cluster bereitgestellt werden. Das PV steht dem gesamten Cluster zur Verfügung und ist keinem Namespace zugeordnet.

Aus technischer Sicht werden die Persistent Volume Objekte im API-Server erstellt. Die Persistent Volumes werden auf die einzelnen Worker Nodes gemappt. Das Kubelet mappt die PVs auf die einzelnen Pods. Sobald dem Knoten Speicher zugewiesen und der Pod gestartet wurde, wird das Persistent Volume auf den Container gemountet.

PVs können entweder manuell durch einen Administrator oder dynamisch durch einen Storage Klassen Controller bereitgestellt werden. Eine Storage-Klasse definiert eine Gruppe von Speichertypen, die von einer dynamischen Provisionierung-Logik verwendet werden, um PVs automatisch zu erstellen. Wenn ein Pod ein PVC anfordert und keine passenden PVs verfügbar sind, wird automatisch ein neues PV erstellt.

Wenn man ein Persistent Volume definiert, hat man eine Wahl zwischen vielen verschiedenen Arten von Persistent Volumes, die innerhalb von Kubernetes bereitgestellt werden können. Dies ist von der angeschlossenen Storage-Infrastruktur abhängig und lässt sich grob in vier Typen aufteilen: Local-, Netzwerk-, Block- und Cloud-Storage.

- Local Disk
- Netzwerk: NFS, azureFIle
- Block: Fibre Channel, iSCSI
- Cloud: z.B. awsElasticBlockStore, azureDisk, gcePersistentDisk

Die Art des verwendeten Storage ist von der Use Case und / oder Infrastruktur abhängig.

Hier finden Sie eine vollständige Liste: https://kubernetes.io/docs/concepts/storage/persistent-volumes/#types-of-persistent-volumes

Persistent Volume Claim

Der Persistent Volume Claim (PVC) ist eine Anforderung des Benutzers an den Cluster, ihm eine bestimmte Menge an Speicherplatz zu gewähren.

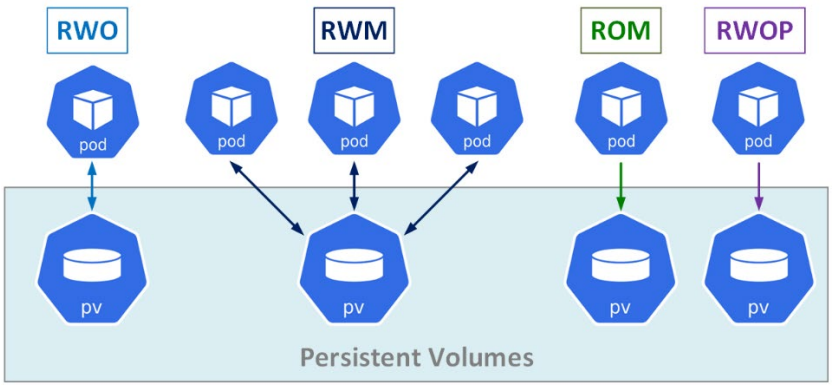
Wenn ein Persistent Volume Claim gestellt wird, müssen dabei eine Reihe von Eigenschaften definiert werden: die Größe des PersistentVolumes, den Zugriffsmodus (Access Mode) für dieses PersistentVolumes, die Speicherklasse (Storage Class) usw. Der Zugriffsmodus definiert, wie das Volume genutzt werden kann.

Wenn ein geeignetes Persistent Volume gefunden wird, das den gestellten Anforderungen entspricht, wird es an den PVC gebunden (Bindung) und bereitgestellt. Wenn kein passendes PV vorhanden, kann in Systemen mit dynamischer Provisionierung (s. unten) automatisch ein neues PV erzeugt werden, das den Anforderungen aus dem PVC entspricht.

PVC-Beispiel

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mein-pvc
  namespace: mein-namespace
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 10Gi
  storageClassName: meine-speicherklasse
```

Access Modes



Persistent Volumes und Persistent Volume Claims stehen vier verschiedenen Zugriffsmodi zur Verfügung:

- RWO – ***ReadWriteOnce*** bedeutet, dass ein Node das Volume sowohl für den Lese- als auch für den Schreib-Zugriff mounten kann.
- RWX – ***ReadWriteMany*** bedeutet, dass mehr als ein Node das Volume für den Lese-Schreib-Zugriff mounten kann.
- ROM – ***ReadOnlyMany*** bedeutet, dass mehr als ein Node das Volume für den reinen Lesezugriff mounten kann.
- RWOP – ***ReadWriteOncePod*** bedeutet, dass ein Volume nur von einem einzelnen Pod im gesamten Cluster mit Lese-Schreib-Zugriff gemountet werden kann. Die Option ist nur ab Version 1.22 und nur für PVC verfügbar.

Es ist zu beachten, dass die zugrundeliegende Speicherkomponente immer noch ihre eigenen Eigenschaften haben kann, die im Widerspruch zu den konfigurierten Einstellungen stehen können.

Static Provisioning

Bei der statischen Bereitstellung wird das Persistent Volume durch einen Administrator vorab erstellt. Dabei legt der Administrator die Spezifikationen (Zugriffsmodi, Größe, Name usw.) fest.

Dynamic Provisioning

Bei einer dynamischen Bereitstellung das Persistent Volume zeitgleich mit der Erstellung von PVC angelegt. Dies passiert in der Regel, wenn die verfügbaren statischen PVs nicht mit der PVC-Spezifikation übereinstimmen. In diesem Fall basiert die Bereitstellung anhand von vordefinierten Storage Klassen.

Storage Class

Das Objekt Storage Class bietet die Möglichkeit bestimmte Eigenschaft wie z.B. Leistung, Größe oder Zugriffsart, sowie die infrastrukturspezifischen Parameter zu definieren. Innerhalb des StorageClass werden auch die Schritte definiert (reclaim policy) was mit einem dynamisch zugewiesenen PersistentVolumes geschehen soll, sobald die PVC gelöscht ist.

Storage Lifecycle

Den gesamten Lebenszyklus der PVs/PVCs können wir in drei Phasen aufteilen: Binding, Using und Reclaim. Hier ist eine grobe Beschreibung der einzelnen Schritte.

Provisioning - die erste Phase ist bereits oben erklärt und beinhaltet eine statische oder eine dynamische Provisionierung.

Binding - ist ein Prozess der Zuordnung eines PersistentVolumeClaim zu einem PersistentVolume für den weiteren Zugriff von einem Pod. Aus technischer Sicht passiert Folgendes: wenn der PersistentVolumeClaim erstellt wird, findet ein *Control Loop* diesen PersistentVolumeClaim und versucht, ein passendes PersistentVolume zu finden. Dies kann entweder statisch oder dynamisch passieren. Wenn der *Control Loop* kein PersistentVolume finden kann, bleibt der Pod, welcher diesen PersistentVolumeClaim verwenden wollte, in dem Pending-Zustand, und zwar so lange bis die passende Ressource verfügbar wird.

Using - Das Volume steht dem Pod während seiner Lebensdauer zur Verfügung.

Reclaiming - in dieser Phase wird festgelegt, was mit dem Volume geschehen soll. Sie haben eine Wahl zwischen zwei möglichen Optionen Retain und Delete.

> **Retain** - dem zugrundeliegenden Storage wird mitteilt, dass der Volume noch manuell zurückgefordert werden kann.
>
> **Delete** - die Phase ist selbst erklärend. Diese Option wird überwiegend in dynamischen Provisioning Szenarien verwendet.

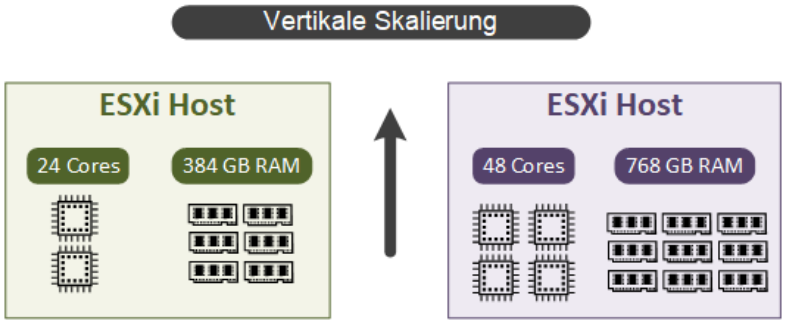
Autoscaling

Was ist Skalierung?

Unter Skalierung versteht man im Allgemeinen die Vergrößerung oder Verkleinerung vorhandener Ressourcen.

Es gibt zwei Arten der Skalierung:

Vertikale Skalierung (Scaling up) - die vertikale Skalierung wird durch das Hinzufügen zusätzlicher Hardware-Ressourcen erreicht. Wie im unteren Bild dargestellt, wurde der vorhandene Server mit mehr CPU und RAM ausgestattet. Auf diese Weise können sowohl physische als auch virtuelle Ressourcen erweitert werden.



Was ist Kubernetes Autoscaling?

Kubernetes Autoscaling ermöglicht eine dynamische Anpassung an steigenden oder sinkenden Ressourcenbedarf durch horizontale und vertikale Skalierung der einzelnen Ressourcen.

Im Gegensatz zum „Legacy“ Scaling orientiert sich Kubernetes Autoscaling nicht an der Skalierung von physischen oder virtuellen Maschinen, sondern muss eine andere

Abstraktionsebene bedienen. Beim Kubernetes Autoscaling liegt der Fokus auf den Applikationen und den architekturbedingt darunter liegenden Pods.

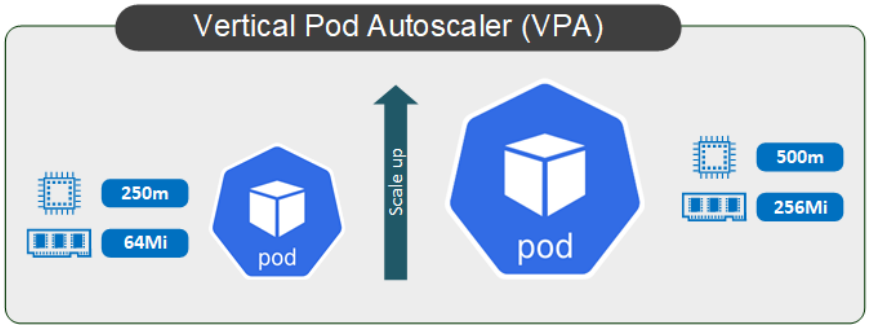
Da die Kubernetes-Infrastruktur hauptsächlich in der Cloud eingesetzt wird, hilft Kubernetes Autoscaling bei der Kostenoptimierung, indem ein Cluster dynamisch nach oben und unten skaliert wird, je nach aktuellem Bedarf.

Autoscaling-Funktionen für Kubernetes

In Kubernetes gibt es drei Autoscaling-Funktionen.

- Vertical Pod Autoscaler (VPA) - erhöht oder reduziert CPU- und Speicher-Ressourcen in Pod
- Horizontal Pod Autoscaler (HPA) - fügt neue Pods hinzu oder entfernt diese
- Cluster Autoscaler - kann Clusterknoten hinzufügen oder entfernen

Vertical Pod Autoscaler (VPA)



Wie bereits erwähnt, basiert VPA auf dem gleichen Prinzip wie die „klassische“ vertikale Skalierung, d.h. das Hinzufügen oder Entfernen von Ressourcen (z.B. CPU oder Speicher) zu oder von einem Pod. Man könnte VPA als eine Weiterentwicklung der in Kubernetes üblichen Requests und Limits (s. unten) betrachten, allerdings mit einem wesentlichen Unterschied: Die Requests werden automatisch auf Basis einer Verhaltensbeobachtung (ca. 5 Minuten) und einer anschließenden Bewertung aktualisiert.

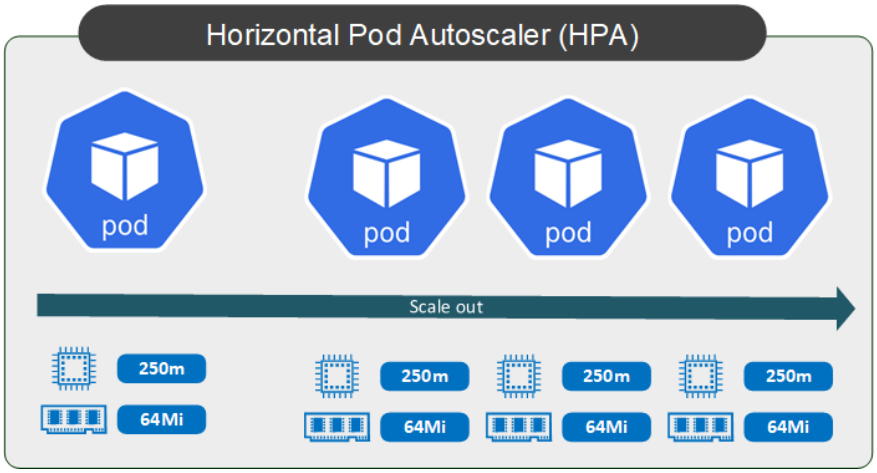
Für die „technische“ Umsetzung ist ein Kubernetes-Objekt namens *VerticalPodAutoscaler* zuständig, zu dem noch drei Komponenten gehören:

Der ***Recommender*** ist die zentrale Komponente des VPA. Der Recommender dient zur Überwachung des historischen und aktuellen Ressourcenverbrauchs und zur Berechnung der daraus resultierenden CPU- und Speicheranforderungen.

Link: https://github.com/kubernetes/autoscaler/blob/master/vertical-pod-autoscaler/pkg/recommender/

Der ***Updater*** ist für die Umsetzung der vom Recommender berechneten Empfehlungen verantwortlich. Der Updater erstellt neue Pods und beendet Pods, die aktualisiert werden müssen. Die eigentliche Aktualisierung wird jedoch vom ***Vertical Pod Autoscaler Admission Plugin*** durchgeführt.

Horizontal Pod Autoscaler (HPA)



Wie auf dem Bild zu sehen ist, unterscheidet sich HPA von VPA durch die Erhöhung oder Reduzierung der Pod-Anzahl und nicht durch die Anpassung deren Eigenschaften.

HPA ist wesentlich flexibler als VPA, da durch das Hinzufügen / Entfernen von Pods eine kosteneffiziente und performante Umgebung gebaut werden kann. Ein weiterer, eher hypothetischer Vorteil ist die Möglichkeit, die Performance-Engpässe im Bereichen Storage I/O und Netzwerk abzufedern. Wobei die Entscheidung weiteren Pods zu starten, wird ausschließlich auf Basis von CPU- und Arbeitsspeicher-Metriken getroffen. Die realen Engpässe bei IOPS, Netzwerk und Storage werden nicht berücksichtigt.

Einzige Voraussetzung für den Einsatz von HPA ist, dass die verwendeten Applikationen unter Berücksichtigung der horizontalen Skalierung entwickelt wurden und die parallele Ausführung mehrerer Instanzen unterstützen.

HPA Komponenten

Bei dem Aufbau vom HPA werden mehreren Komponenten zum Einsatz kommen: HPA, cAdvisor, Metrics API, API Service und Mertics Server.

Auf jedem Node ist eine Komponente namens **cAdvisor** eingebaut (als Teil von kubelet) und dient zur Überwachung der Ressourcenauslastung der laufenden Container. cAdvisor sammelt mehrere interne Metriken in einem Node, die aber von keinem Tool genutzt werden.

Link: https://github.com/google/cadvisor

Die vom cAdvisor gesammelten Metriken werden von einem weiteren Tool namens **Mertics Server** aggregiert. Der Metrics Server wird über die Metrics API für HPA und VPA zur Verwendung bereitgestellt.

Link: https://github.com/kubernetes/metrics

Der **Metrics Server** ist eine separate Komponente und wird bei der Cluster-Installation nicht mitinstalliert. Für die Installation müssen einige Voraussetzungen erfüllt sein. Der Metrics Server läuft als Pod im Kubernetes-Cluster und sammelt in regelmäßigen Abständen (standardmäßig 60 Sekunden) die Metriken (CPU und Speicher) der Nodes. Die gesammelten Metriken werden nicht aufbewahrt und sind zur sofortigen Verwendung bestimmt.

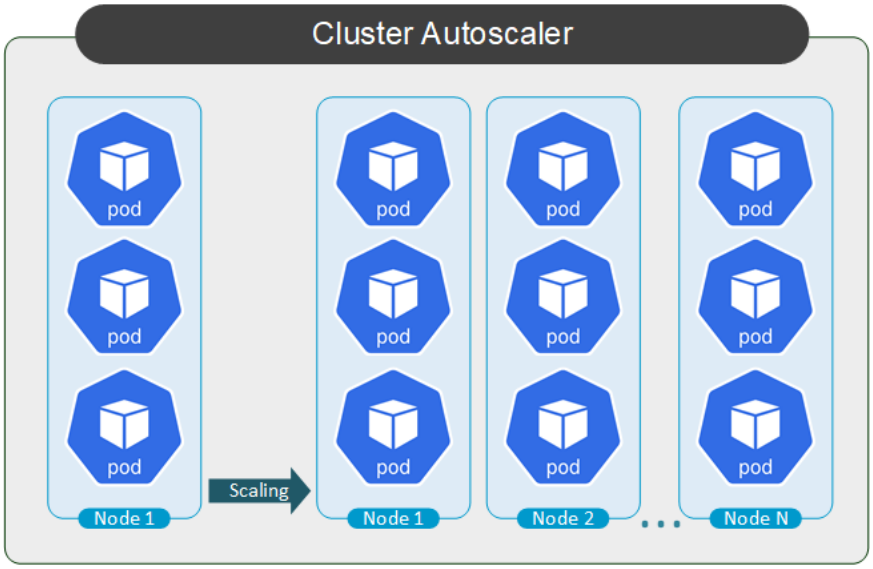
Link: https://github.com/kubernetes-sigs/metrics-server

HPA-Versionen

Es gibt zwei HPA-Versionen: v1 und v2. Die erste HPA-Version (autoscaling/v1) hat eine sehr begrenzte Konfigurationsmöglichkeit und ist ausschließlich auf der durchschnittlichen CPU-Auslastung basiert.

Die zweite Version (autoscaling/V2beta1 und autoscaling/V2beta2) unterstützt die Verwendung von mehreren Metriken. Diese Metriken können auch vom Benutzer definiert oder aus externen Quellen stammen.

Cluster Autoscaler



Cluster Autoscaler ermöglicht die automatische Skalierung des Clusters selbst, indem die Anzahl der Knoten erhöht oder verringert wird.

Es kann die Situation eintreten, dass trotz dynamischer Ressourcenverteilung alle zugewiesenen Ressourcen erschöpft sind und keine weiteren Pods auf den vorhandenen Worker Nodes gestartet werden können. In diesem Fall gibt es nur eine Möglichkeit, bestehende Engpässe zu beseitigen: die Anzahl der Worker Nodes zu erhöhen.

Im Gegensatz zu den beiden anderen Skalierungsmethoden (VPA/HPA) ist der Cluster Autoscaler kein Bestandteil des Kubernetes Clusters, da Kubernetes keine Mechanismen für das automatische Anlegen und Entfernen von virtuellen Maschinen besitzt. Der Cluster Autoscaler ist eine „typische“ Komponente von Managed Kubernetes bei Cloud Providern.

Requests und Limits

Die Requests / Limits sind optionale Konfigurationsmöglichkeiten, die zur Optimierung der Ressourcennutzung verwendet werden können. Requests / Limits müssen nicht konfiguriert werden. Je nach Anwendung oder Situation können die Requests / Limits die Stabilität der einen oder anderen Anwendung positiv oder negativ beeinflussen. Diese Einstellung ist als zweischneidiges Schwert zu betrachten.

Requests

Die Requests dienen zwei Zwecken, erstens einen geeigneten Knoten mit ausreichender Kapazität zu finden und zweitens die Gesamtmenge der benötigten Ressourcen zu berechnen.

Die Requests definieren die minimale Menge an RAM/CPU, die für den Container benötigt wird. Wie bereits erwähnt, entscheidet der Kube Scheduler anhand dieser Informationen, auf welchem Worker Node der Pod gestartet werden soll und reserviert die angeforderte Menge an Ressourcen für diesen Container, so dass diese garantiert zur Verfügung stehen.

Wenn auf keinem einzigen Worker Node die verlangten Ressourcen vorhanden sind, wird der Pod erstmal in einen Pending-Zustand versetzt. Erst wenn die angefragten Ressourcen wieder vorhanden sind, wird der Pod ausgeführt.

Limits

Die Limits sorgen dafür, dass der Container seinen CPU- oder RAM-Verbrauch in Grenzen hält und keine zusätzlichen Ressourcen als vordefiniert, für sich beansprucht, auch wenn solche Ressourcen physikalisch vorhanden sind.

Wenn die Ressource die durch die Limits festgelegte Grenze erreicht, werden bestimmte Ereignisse ausgelöst, die unterschiedliche Auswirkungen auf die CPU und den Speicher haben. CPU-Throttling und Out-of-Memory-Kills sind die bekanntesten.

Hier ein kurzes Beispiel dafür, wie Sie Requests und Limits für eine Container-Spezifikation festlegen können:

```
apiVersion: v1
kind: Pod
metadata:
  name: busybox-app
spec:
  containers:
  - name: busybox
    image: busybox
    resources:
      requests:
        memory: "128Mi"
        cpu: "250m"
      limits:
```

memory: "256Mi"
cpu: "500m"

Einheiten

Die CPU-Anforderungen werden in Millicores (auch milliCPU genannt) („m“) oder in Cores festgelegt, wobei 1000 Millicores = 1 vCPU oder 1 physische CPU Core entspricht. Der minimale Wert beträgt 0.1 Core.

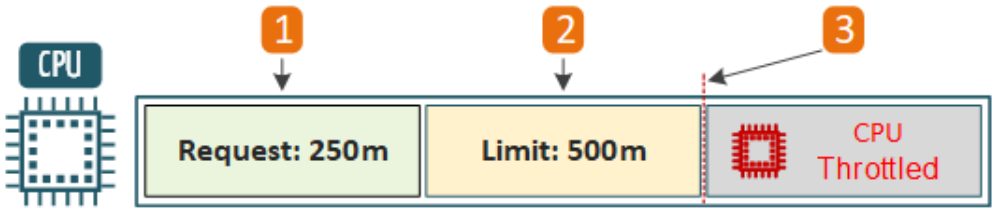
250m auf dem oberen Bespiel bedeutet, dass die ¼ einer CPU aufgefordert wurde und die Limitierung auf 500m gesetzt ist.

Die Memory-Anforderungen werden in Bytes gemessen. Für die Eingabe der Speicher-Ressourcen können unterschiedlichen Suffixe (einstellige und zweistellige) verwendet werden (z.B. Mi bedeutend Mebibyte und entspricht 1,04858 MB, oder 128 MebiByte ist gleich zu 135 Megabyte (MB)).

CPU-Throttling

CPU-Throttling bedeutet, dass die CPU-Nutzung eines Containers gedrosselt wird, wenn er sein CPU-Limit überschreitet. Das CPU-Throttling wird durch sogenannte "Throttling-Perioden" geregelt. Die "Throttling-Periode" ist ein Mechanismus zur Überprüfung der CPU-Nutzung des Containers in regelmäßigen Abständen (standardmäßig 100ms). Aus technischer Sicht sind die cgroups (Linux-Funktionen) die CPU-Drosselung zuständig.

Wenn z.B. ein Container ein Limit von "500m" hat, was 50% eines CPU-Kerns entspricht. Das bedeutet, dass dieser Container in jeder Throttling-Periode (100ms) bis zu 50ms CPU-Zeit verbrauchen (ebenfalls 50%) darf. Hätte ein Container ein Limit von "300m" (30% eines CPU-Kerns), so hätte er Anspruch auf 30ms CPU-Zeit.



1. Für einen Container reservierter CPU-Anteil.
2. Maximale CPU-Auslastung für den Container.
3. Grenzwert, bei dem das CPU-Throttling beginnt.

Die potentiellen Probleme des CPU-Throttling lassen sich anhand des folgenden Beispiels noch besser verdeutlichen:

Eine Funktion einer Applikation benötigt normalerweise 80ms CPU-Zeit, um ihre Tasks abzuschließen.

- CPU-Limit: 500m
- Throttling-Periode: 100ms

- Verfügbare CPU-Zeit pro Periode: 50ms (50%)

Ohne CPU-Limit hätte der Container die 80ms CPU-Zeit-Aufgabe direkt abgearbeitet.

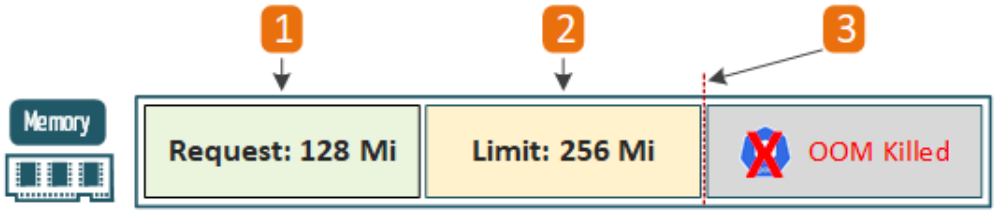
Andernfalls werden die CPU-Limits wie folgt auswirken:

- Erste Periode (100ms): 50ms Arbeitszeit und wird dann gedrosselt.
- Zweite Periode (nächste 100ms): die restlichen 30ms werden abgearbeitet.

Die CPU-Limits können zu Leistungseinbußen und längeren Antwortzeiten der Anwendung führen. Je mehr Iterationen eine Anwendung benötigt, desto schlechter sind die Performance-Werte. Dabei spielt es keine Rolle, ob die CPU des Nodes grundsätzlich ausgelastet ist oder nicht.

Out-Of-Memory (OOM)

Wenn ein Container mehr Speicher (RAM) anfordert, als ihm durch Limit zur Verfügung steht, wird er durch den OOM-Killer beendet und neu gestartet. Solche unerwarteten Neustarts können zu Instabilität und Datenverlust der Anwendung führen.



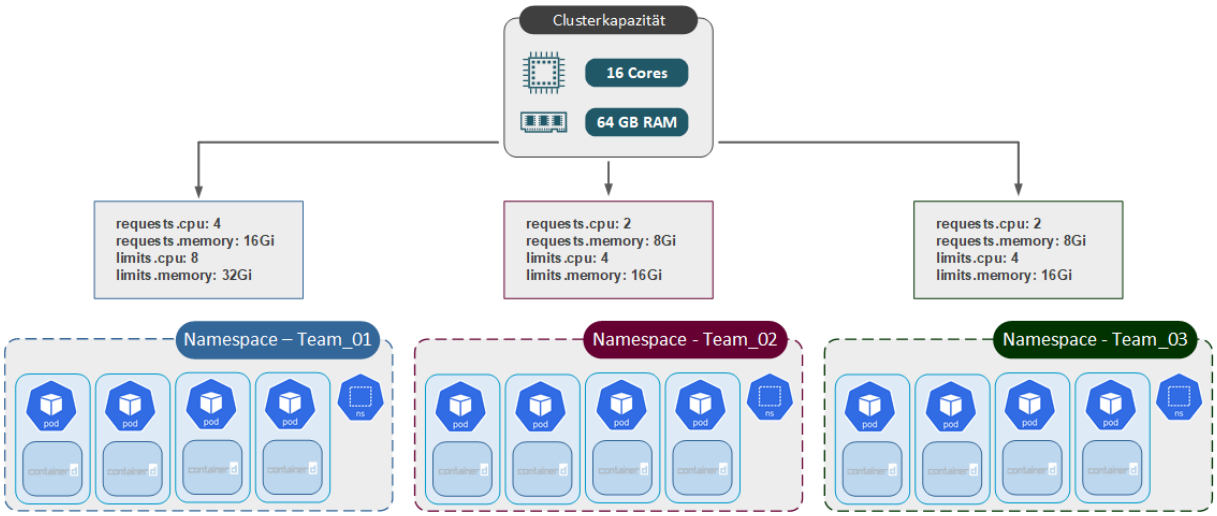
1. Maximale Menge an Memory, die für den Container reserviert ist
2. Maximaler Memory-Verbrauch für den
3. Out-of-Memory greift ein und startet den Pod

Resource Quotas

Die Verwendung von Resource Quotas bietet die Möglichkeit, die Nutzung der CPU- und Speicher-Ressourcen auf Namespace-Ebene einzuschränken. Diese Technologie (Kubernetes Objekt: ResourceQuota) ist für den UseCase vorgesehen, wenn unterschiedlichen Teams einen Kubernetes Cluster teilen. So wird die angemessene Verteilung ermöglicht. Grundsätzlich können Sie mit Resource Quotas auch die Nutzung den anderen Objekten verwenden. Dies wird aber in diesem Betrag nicht behandelt.

Hier ist ein Bespiel der ResourceQuotas- Konfiguration, welche für jeden Namespace erstellt werden muss:

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: team-alpha
spec:
  hard:
    requests.cpu: 2
    requests.memory: 2Gi
    limits.cpu: 4
    limits.memory: 4Gi
    requests.cpu: 2 -
    requests.memory: 2Gi
    limits.cpu: 4
    limits.memory: 4Gi
```



requests.cpu / requests.memory - die Summe aller CPU- / Sprecher-Anforderungen darf nicht höher sein als der hier definierte Wert.

limits.cpu / limits.memory - die hier eingegebene Grenzwerte dürfen nicht überschritten werden.

Falls die Gesamtkapazität eines Clusters kleiner ist als die Summe aller Kontingente der Namespaces, kann es zu Ressourcenkonflikten kommen, die nach Prinzip „First Come First Serve“ gelöst werden.

Die Verwendung von ResourceQuota kann sowohl gemeinsam mit den Requests/Limits auf der Pod-Ebene als auch separat genutzt werden.

Quality of Service (QoS)

In Kubernetes gibt es drei Arten von Quality of Service (QoS) Klassen für Pods, die einen direkten Bezug zu unserem Thema haben.

- Guaranteed
- Burstable
- BestEffort

Die Zuordnung der Pods zu einer oder anderen QoS-Klasse wird durch die definierten Requests und Limits sowohl für die CPUs als auch für den Speicher bestimmt.

Guaranteed

Die Zuordnung zu dieser Klasse erfolgt nur, wenn **für jeden Container** sowohl Requests als auch Limits für CPU/Speicher konfiguriert sind. Die Werte der Requests müssen mit den Limits übereinstimmen. Wenn dies der Fall ist, wird diesen Pods die höchste Priorität bei der Ressourcenzuteilung garantiert.

Burstable

Zu dieser Klasse gehören Pods, bei denen **mindestens ein Container** ungleiche Request oder Limits für CPU/Speicher hat, oder bei denen Request, aber keine Limits gesetzt sind. CPU Throttling kann für Container mit einem definierten Limit angewendet werden. Diese Pods haben eine mittlere Priorität und können zusätzliche Ressourcen nutzen, aber nur wenn diese verfügbar sind.

BestEffort

BestEffort ist das rechtlose Mitglied der QoS-Klassen. Pods, für die weder Requests noch Limits (für CPU und Speicher) definiert sind, werden der Klasse BestEffort zugeordnet. Diese Pods haben die niedrigste Priorität und werden CPU-mäßig als die letzten bedient und sind die ersten Kandidaten, die vom OOM Killer des Betriebssystems beendet werden.

Secrets

Kubernetes Secrets ist eine Ressource in Kubernetes, die dient zur Speicherung von vertraulichen Informationen wie Passwörtern, Schlüsseln und Zertifikaten verwendet wird. Secrets können von Pods, Containern oder anderen Kubernetes-Objekten wie Deployments und Services verwendet werden, ohne dass die vertraulichen Informationen direkt im YAML-Manifest gespeichert werden müssen.

Die Verschlüsselung der Kubernetes Secrets erfolgt intern in Kubernetes und benötigt keine externe Komponente. Die Secrets werden standardmäßig in base64 kodiert (s. Base64 Sicherheitsbedenken), was jedoch keine Verschlüsselung darstellt, sondern nur eine einfache Kodierung. Die Kubernetes Secrets werden im etcd-Cluster gespeichert und vom API-Server verwaltet.

Secrets können in Pods über Umgebungsvariablen oder Volumes gemountet werden. Wenn Secrets als Umgebungsvariablen verwendet werden, werden die vertraulichen Informationen in Umgebungsvariablen im Pod ausgeführt. Wenn Secrets als Volumes verwendet werden, werden die vertraulichen Informationen in einer Datei gespeichert, die im Pod als Volumes gemountet ist.

Secrets können auch mit Labels und Annotations versehen werden, um eine einfache Suche und Organisation zu ermöglichen. Secrets können auch aktualisiert und gelöscht werden, um sicherzustellen, dass vertrauliche Informationen auf dem neuesten Stand und sicher bleiben.

Offizielle Dokumentation: Secrets | Kubernetes

Arten von Secrets

Es gibt zwei Arten von Secrets in Kubernetes: generische Secrets und Secrets für Image-Repositories (Container-Registry). Generische Secrets werden für allgemeine vertrauliche Informationen wie Passwörter und Schlüssel verwendet, während Secrets für Image-Repositories verwendet werden, um Anmeldeinformationen für die Verbindung zu einer Container-Registry zu speichern. In diesem Fall werden Secrets verwendet, um den Benutzername und Passwort oder Token in einem Kubernetes-Cluster zu speichern, damit Pods oder Deployments auf das private Repository zugreifen können.

Speicherung der Secrets

Folgende Möglichkeiten können verwendet werden, um eine sichere Speicherung der Secrets zu gewährleisten:

Secret Encryption Config

In diesem Fall werden die Secrets mit einem symmetrischen Schlüssel verschlüsselt, der in einem separaten Secret gespeichert wird. Dies erhöht die Sicherheit, da der Schlüssel selbst verschlüsselt ist und nur von autorisierten Benutzern entschlüsselt werden kann.

Verwendung von Secrets-Management-Tools

Durch die Verwendung von Vault können Kubernetes-Cluster einheitliche Methoden zur Verwaltung von Secrets implementieren und die Sicherheit von Secrets erhöhen. Wenn von Vault die Rede ist, ist eigentlich HashiCorp Vault gemeint.

HashiCorp Vault ist eine zentrale Plattform zur Verwaltung von Secrets, wie z.B. Passwörtern, API-Schlüsseln, Tokens und Zertifikate, sowie zur sicheren Generierung von Zufallszahlen und -werten.

Die Secrets werden in Vault in sogenannten "Secret Engines" gespeichert, die je nach Bedarf konfiguriert werden können. Beispielsweise kann ein Secret Engine für Passwörter und ein anderer für Tokens erstellt werden. Für jede Art von Secret gibt es einen eindeutigen Pfad, unter dem die Secrets innerhalb des Secret Engines abgelegt werden.

Base64 Sicherheitsbedenken

Die Base64-Kodierung ist ein Verfahren zur Kodierung von Binärdaten in Textdaten und umgekehrt. Die Codierung ist keine Verschlüsselung und kann relativ einfach rückgängig gemacht werden. Das bedeutet, dass jemand, der Zugriff auf die Secret-Datei hat, die Base64-kodierten Informationen ohne viel Aufwand entschlüsseln kann. Es ist daher wichtig, sensible Informationen wie Passwörter und Zugangsdaten in Kubernetes Secrets durch Verschlüsselung zu schützen, bevor sie gespeichert oder übertragen werden.

KCNA – Prüfung

Die KCNA (Kubernetes and Cloud Native Associate) Prüfung ist eine Zertifizierung, die von der Linux Foundation angeboten wird. Es werden Grundkenntnisse der Kubernetes-Technologie und der Kubernetes-Cluster-Architektur abgefragt.

Inhalt der Zertifizierung (offizielle Information)

- Die Zertifizierung bestätigt konzeptionelles Wissen über das gesamte Cloud-Native-Ökosystem, insbesondere über Kubernetes.
- Sie bereitet Kandidaten darauf vor, mit Cloud-Native-Technologien zu arbeiten und weitere CNCF-Zertifizierungen wie CKA, CKAD und CKS anzustreben.

Kompetenzbereiche (offizielle Information)

- Kubernetes-Grundlagen (46%): Ressourcen, Architektur, API, Container, Scheduling.
- Container-Orchestrierung (22%): Grundlagen, Laufzeit, Sicherheit, Netzwerk, Service Mesh, Speicher.
- Cloud-Native-Architektur (16%): Autoscaling, Serverless, Community und Governance.
- Cloud-Native-Beobachtbarkeit (8%): Telemetrie, Observability, Prometheus, Kostenmanagement.
- Cloud-Native-Anwendungsbereitstellung (8%): Grundlagen, GitOps, CI/CD.

Prüfungsdetails

- Es gibt keine spezifischen Voraussetzungen
- Die Prüfung ist ein online überwachtes, Multiple-Choice-Examen.
- Die Zertifizierung ist drei Jahre gültig
- Die Prüfungsdauer beträgt 90 Minuten
- Der Preis für die Prüfung beträgt 250 USD

Offizielle Information:

https://training.linuxfoundation.org/certification/kubernetes-cloud-native-associate/

https://docs.linuxfoundation.org/tc-docs/certification/frequently-asked-questions-kcna