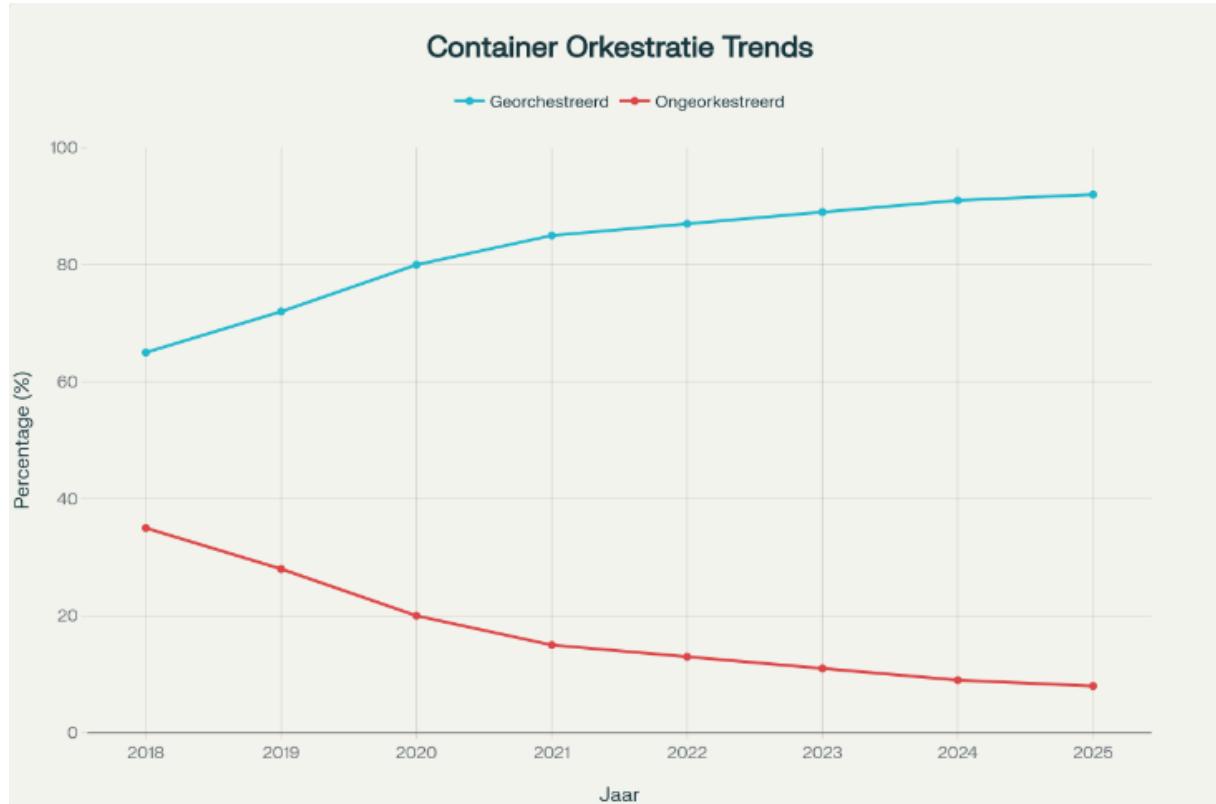


11 Kubernetes

11.1 Inleiding

De tijd dat we manueel containers moeten starten en stoppen is stilaan voorbij. Tegenwoordig worden de meeste containers (voornamelijk in de grotere bedrijven) beheerd door een orchestration tool.

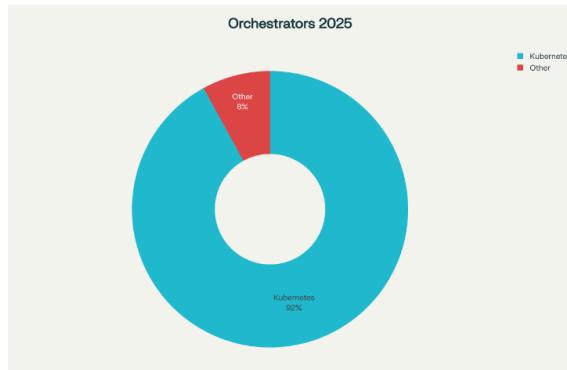


Kubernetes is het belangrijkste open-source platform om containergebaseerde applicaties automatisch te deployen, te schalen en te beheren. Het zorgt ervoor dat applicaties betrouwbaar draaien op een cluster van computers, ongeacht waar ze worden gehost (zowel in de cloud als on-premise).

Kubernetes kan bijvoorbeeld:

- Containers starten en stoppen
- Schaalvergroting of -verkleining op basis van vraag
- Load balancing
- Zelfherstel bij fouten

Het helpt om applicaties stabiel en efficiënt te draaien zonder handmatig beheer van servers. Wereldwijd wordt Kubernetes aanschouwd als de beste tool voor Container Orchestration.



Enkele alternatieven op een rij:

- Docker Swarm (met Mirantis-ondersteuning)
Eenvoudiger te beheren, maar minder schaalbare orchestrator dan Kubernetes.
- Amazon ECS
De managed container orchestrator van AWS, vooral populair bij wie volledig in AWS-cloud werkt.
- OpenShift
Gebaseerd op Kubernetes, ontwikkeld door RedHat; wordt als compleet platform beschouwd met extra features voor ontwikkelaars en security.

11.2 Kubernetes architectuur

11.2.1 Clusters

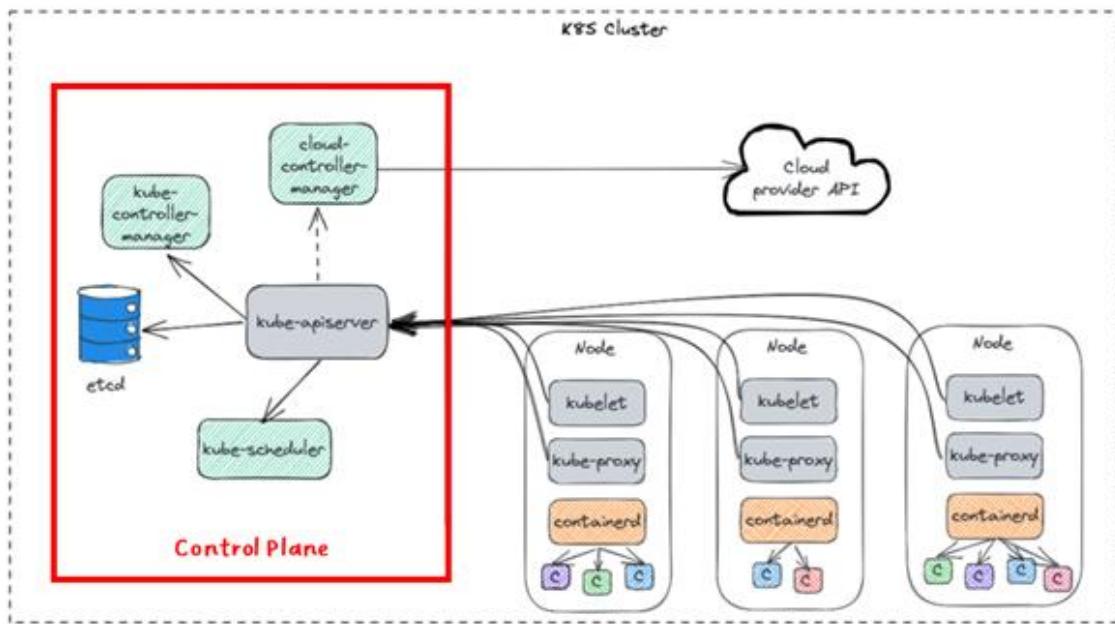
Binnen Kubernetes wordt met een cluster een groep machines (hosts) bedoeld die samenwerken om containerized applicaties te draaien en te beheren. Er zijn bij Kubernetes verschillende hosts die samenwerken die één bepaalde functie hebben (worker node of control plane) om een applicatie te hosten.

Binnen een Kubernetes-cluster wordt een host een "node" genoemd. Een node kan een fysieke machine zijn, maar ook een virtuele machine, bijvoorbeeld in een cloudomgeving.

Er zijn verschillende cloudproviders die Kubernetes als beheerde dienst aanbieden. Om echter een goed begrip te krijgen en te leren werken met Kubernetes, beperken we ons tot het gebruik van lokale virtuele nodes.

Naast de gewone worker nodes, is er ook de control plane node. Waar de worker nodes de uitvoerders zijn van de taken binnen het cluster, fungeert de control plane als de beheerder die het cluster coördineert en controleert. Voordat we beginnen met het opzetten van een Kubernetes-cluster, is het belangrijk om beide componenten goed te begrijpen.

Hieronder vind je de architectuur terug van Kubernetes.



11.2.2 Control Plane

Het brein van een Kubernetes-cluster is de Control Plane node. Dit is een verzameling van verschillende services die samenwerken om de volledige cluster correct te laten functioneren.

In een productieomgeving draait een Control Plane node nooit alleen. Voor voldoende betrouwbaarheid en beschikbaarheid worden altijd meerdere Control Plane nodes ingezet, vaak in een High Availability (HA) configuratie met minstens 3 of 5 nodes. Dit zorgt ervoor dat het cluster blijft werken, zelfs als één of meerdere Control Plane nodes uitvallen.

Voor test- en ontwikkelomgevingen volstaat meestal één enkele Control Plane node. Maar in een productieomgeving is het aan te raden om altijd minimaal drie Control Plane nodes te gebruiken om redundantie en fouttolerantie te garanderen.

Kortom, de Control Plane beheert het cluster, verzorgt planning, statuscontrole, en coördinatie, en draait in HA setups meerdere keren om continuïteit te waarborgen.

Een Control Plane node in Kubernetes bestaat uit verschillende belangrijke services, die elk een specifieke taak binnen het cluster hebben. Zonder diep op alle details in te gaan, volgt hier een korte uitleg van de belangrijkste componenten:

- API server

De API-server is het centrale aanspreekpunt waarmee alle andere nodes binnen een Kubernetes-cluster communiceren met het control plane.

De API server luistert standaard op poort 6443 en ontvangt alle verzoeken binnen het cluster.

- Cluster Store (etcd)

Dit is de opslagplaats waarin alle configuraties en de huidige status van het gehele cluster worden bewaard.

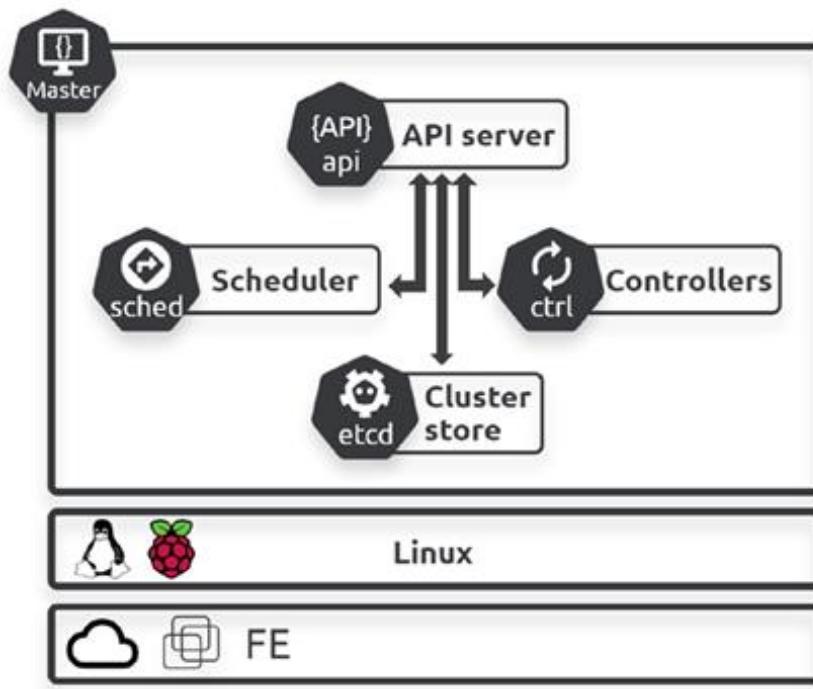
Het functioneert als een consistente en betrouwbare database.
- Controller Manager

Dit is een verzameling controllers die verschillende aspecten van het cluster beheren. De hoofdtaak is om ervoor te zorgen dat de werkelijke toestand van het cluster altijd overeenkomt met de gewenste toestand, bijvoorbeeld dat de juiste applicaties draaien zoals geconfigureerd.
- Scheduler

De scheduler is verantwoordelijk voor het toewijzen van nieuwe taken aan gezonde nodes. Hij beslist waar nieuwe pods moeten worden gestart door de status van de nodes te controleren via de API server.
- Cloud Controller Manager

Deze service maakt het mogelijk om het Kubernetes-cluster te integreren met cloudproviders. Het biedt een centraal punt voor communicatie en beheer van cloudspecifieke onderdelen zoals load balancers en opslag.

Deze services vormen samen het Control Plane en zorgen ervoor dat het Kubernetes-cluster betrouwbaar, schaalbaar en automatisch beheerd wordt.



Uiteraard zal je nooit user applicaties uitvoeren op je Control Plane Node.

11.2.3 Worker nodes

Worker nodes zijn de primaire werkpaarden van een Kubernetes-cluster. Zij zijn verantwoordelijk voor het hosten en uitvoeren van de applicaties.

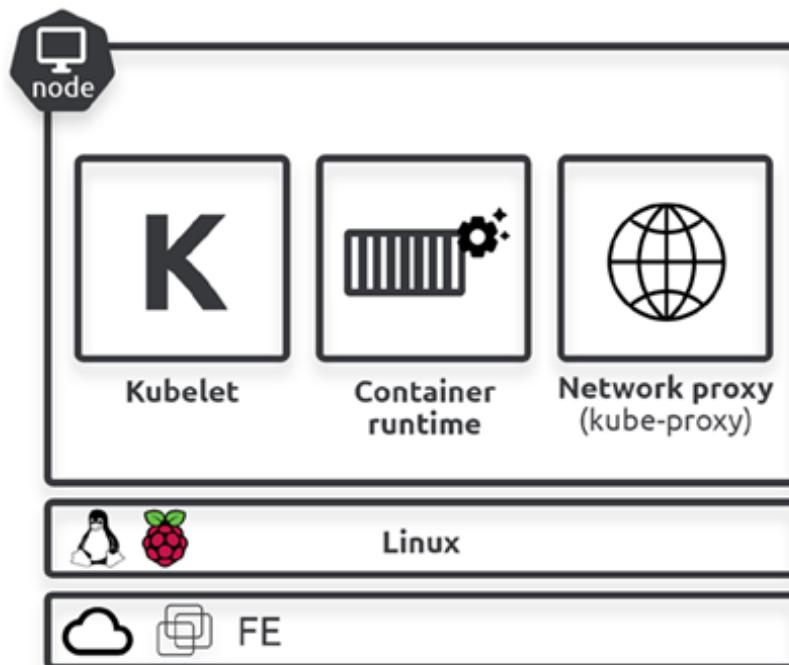
De belangrijkste taken van een worker node zijn:

- Continu de Control Plane raadplegen via de API-server voor nieuwe taken (work assignments).
- Uitvoeren van toegewezen taken zodra deze beschikbaar zijn.
- Regelmatisch rapporteren aan de Control Plane over hun status.

Een worker node bestaat uit verschillende essentiële componenten:

- Kubelet
Een agent die ervoor zorgt dat de containers binnen pods correct draaien en onderhoudt de communicatie met de Control Plane.
- Container runtime
De software die containers uitvoert, zoals containerd of CRI-O.
- Kube-proxy
Verantwoordelijk voor de netwerkregels en het afhandelen van netwerkverkeer naar de containers binnen de node.

Deze taken worden continu herhaald zolang de node actief is, waardoor de applicaties betrouwbaar en schaalbaar kunnen draaien binnen het cluster.



Laat ons eerst even kort ingaan op deze verschillende onderdelen. In tegenstelling tot de Control Plane, gaan we 2 van de 3 onderdelen van een Worker Node uitgebreider bestuderen. De Network-Proxy service gaan we in een later verder uitleggen.

11.2.3.1 Kubelet

De Kubelet is het belangrijkste onderdeel van een worker node in Kubernetes.

Wanneer een worker node aan het cluster wordt toegevoegd, registreert de Kubelet de node bij de API-server van de Control Plane. Daarna onderhoudt de Kubelet continu verbinding met de API-server om te controleren of er nieuwe taken (pods) aan die node zijn toegewezen.

Wanneer er een nieuwe taak is, zorgt de Kubelet ervoor dat deze wordt uitgevoerd door een container te starten. Daarnaast onderhoudt de Kubelet een voortdurende communicatie met de Control Plane om voortdurend statusupdates te versturen over de uitvoering van de taak.

Als de Kubelet een taak niet kan uitvoeren, rapporteert hij dit aan de Control Plane, die dan beslist wat de volgende stap is, bijvoorbeeld om de taak toe te wijzen aan een andere node. De Kubelet wordt automatisch geïnstalleerd en geactiveerd zodra een node zich bij het cluster aansluit, en speelt een cruciale rol in het draaiende houden van containerized applicaties binnen Kubernetes.

11.2.3.2 Container Runtime

Kubernetes zorgt voor de orkestratie van containers, wat betekent dat de nodes in staat moeten zijn om containers uit te voeren. Daarom is er een Container Runtime nodig op elke node. De Container Runtime haalt de benodigde container images op en start of stopt containers op de node op basis van de taken die de Kubelet ontvangt.

Vroeger had Kubernetes native ondersteuning voor Docker als container runtime. Inmiddels maakt Kubernetes gebruik van containerd, een lichtgewicht container runtime die is ontstaan uit de kerncomponenten van Docker. Deze container runtime, containerd, is door Docker Inc. aan de CNCF (Cloud Native Computing Foundation, de organisatie achter Kubernetes) gedoneerd en wordt nu universeel ondersteund in Kubernetes-omgevingen.

In plaats van containerd kan je ook gebruik maken van CRI-O.

Voor pure Kubernetes-gebruik is CRI-O een gespecialiseerde, lichte en veilige runtime. Containerd biedt iets meer functionaliteit en bredere compatibiliteit en is momenteel iets populairder in cloudgestuurde en algemene Kubernetes-omgevingen

11.3 Installatie

11.3.1 k3s versus k8s

Kubernetes draait doorgaans op clusters van meerdere machines (nodes), waarbij er een control plane en meerdere worker nodes zijn, die samen containerized applicaties beheren. Dit type architectuur is ideaal voor productieomgevingen die schaalbaarheid, hoge beschikbaarheid en veerkracht vereisen.

Nu we de basisarchitectuur van een K8's cluster onder de loep hebben genomen, is het tijd om zelf onze eerste cluster op te zetten. We zullen gebruik maken van K3s.

K3s is een lichtgewicht, compacte Kubernetes-distributie ontworpen voor o.a. ontwikkel- en testomgevingen. Het draait Kubernetes functionaliteit op één machine (single-node cluster) of een kleine cluster van machines met minimale resources.

In k3s kunnen control plane en worker nodes gecombineerd binnen dezelfde node. Dit maakt k3s ideaal om snel een complete maar eenvoudige Kubernetes-omgeving lokaal of in resourcebeperkte situaties op te zetten. K3s kan ook in een multi-node setup draaien maar dat zullen we niet bespreken.

Het draaien van een Kubernetes-cluster op één machine heeft vooral zin voor:

- Leer- en testdoeleinden
Het is veel eenvoudiger en sneller om Kubernetes te leren en te experimenteren op één machine zonder meerdere fysieke servers of complexe infrastructuur op te zetten.
- Ontwikkelomgevingen
Developers kunnen lokaal applicaties in containers draaien en testen met dezelfde orkestratiertools die in productie gebruikt worden, wat consistentie bevordert.
- Kleine productieomgevingen
Voor kleine bedrijven of specifieke workloads die geen volledige schaalbaarheid of hoge beschikbaarheid nodig hebben, biedt zo'n setup toch veel voordelen van container orkestratie.

Het opzetten van K8s valt buiten de doelstellingen van de cursus.

11.3.2 VM

Maak een nieuwe RHEL 10 VM aan die aan volgende eisen voldoet:

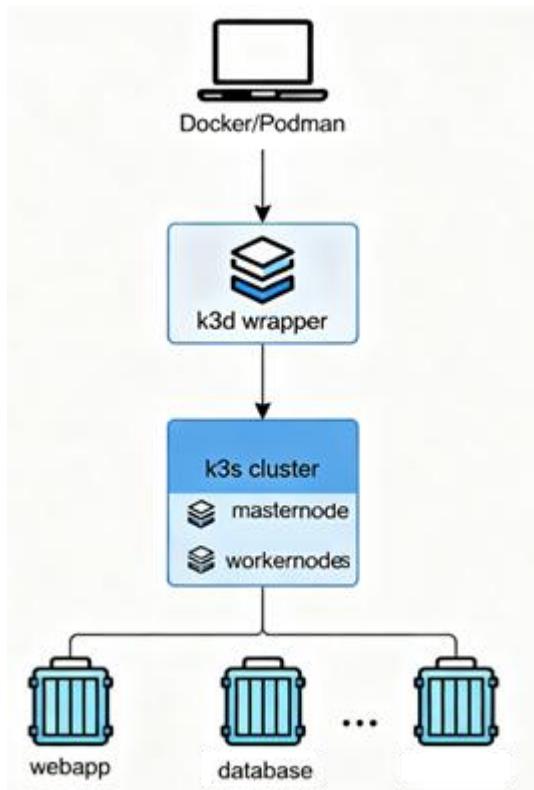
- Geen GUI.
- Hostname: virt-K8s-XX (xx zijn je initialen).
- Gebruiker student met wachtwoord PXL.
- Statisch IP: 192.168.112.10.
- Bereikbaar is via SSH.

11.3.3 Installatie Kubernetes

11.3.3.1 Mogelijkheden

Zoals reeds aangehaald laat k3s ons toe om de basis van Kubernetes onder de knie te krijgen.

Voor deze cursus gaan we werken met k3d. Dit is een wrapper die k3s draait in docker of podman.



Installatieprocedures verschillen per versie van Red Hat Enterprise Linux (RHEL). Op RHEL 10 ondersteunt Red Hat geen Docker meer (je kan wel via een alternatieve repository Docker installeren zoals we geleerd hebben) en promoot het Podman als container runtime.

Hoewel OpenShift van Red Hat een Kubernetes-platform biedt, is het voorlopig niet wijdverspreid.

Wij focussen ons op k3d op RHEL 10 met Docker maar bieden ook installatierichtlijnen voor k3d met Podman op RHEL 10 en voor k3d met Docker op RHEL 9.

Wil je experimenteren met Podman, dan is Minikube een betere keuze dan k3d, omdat het breder en stabiever is in ondersteuning voor diverse container runtimes, inclusief Podman. Zo heb je een soepelere en betrouwbare Kubernetes-ervaring tijdens het leren en uitproberen.

11.3.3.2 RHEL 9

- Docker installeren.
`student@virt-K8s-XX :~$ sudo dnf config-manager --add-repo https://download.docker.com/linux/centos/docker-ce.repo`
`...`
`student@virt-K8s-XX :~$ sudo dnf install -y docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin`
`...`
`student@virt-K8s-XX :~$ sudo systemctl enable --now docker`
- K3d installeren.
`[student@virt-K8s-XX ~]$ sudo wget -q -O - https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash`

```
Preparing to install k3d into /usr/local/bin
k3d installed into /usr/local/bin/k3d
Run 'k3d --help' to see what you can do with it.
```

- k3d heeft toegang nodig tot de Docker-daemon (docker.sock) om containers te kunnen starten en het k3s-cluster op te zetten. Om als niet-root gebruiker Docker-commando's te kunnen uitvoeren, moet je lid zijn van de dockergroep. Het commando newgrp docker zorgt ervoor dat je huidige shell-sessie overschakelt naar de groep docker zonder dat je uit- en weer hoeft in te loggen.

```
[student@virt-K8s-XX ~]$ sudo usermod -aG docker $USER
[student@virt-K8s-XX ~]$ newgrp docker
```

- Kubectl installeren.
Hoewel k3d het cluster opzet, moet je nog steeds iets hebben om ertegen te praten.

```
[student@virt-K8s-XX ~]$ sudo cat <<EOF | sudo tee
/etc/yum.repos.d/kubernetes.repo
[Kubernetes]
name=Kubernetes
baseurl=https://pkgs.K8s.io/core:/stable:/v1.32/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.K8s.io/core:/stable:/v1.32/rpm/repo
md.xml.key
EOF
[student@virt-K8s-XX ~]$ sudo dnf update && sudo dnf install
kubectl -y
```

11.3.3.3 RHEL 10

Volg onderstaande stappen:

- Docker installeren (zoals beschreven in hoofdstuk 2). De stappen staan hieronder ter herhaling.

```
student@virt-K8s-XX :~$ sudo dnf config-manager --add-repo
https://download.docker.com/linux/centos/docker-ce.repo
...
student@virt-K8s-XX :~$ sudo dnf install -y docker-ce docker-ce-
cli containerd.io docker-buildx-plugin docker-compose-plugin
...
student@virt-K8s-XX :~$ sudo systemctl enable --now docker
```

- k3d heeft toegang nodig tot de Docker-daemon (docker.sock) om containers te kunnen starten en het k3s-cluster op te zetten. Om als niet-root gebruiker Docker-commando's te kunnen uitvoeren, moet je lid zijn van de dockergroep. Het commando newgrp docker zorgt ervoor dat je huidige shell-sessie overschakelt naar de groep.

```
student@virt-K8s-XX :~$ sudo usermod -aG docker $USER
student@virt-K8s-XX :~$ newgrp docker
```

- K3D installeren.

```
student@virt-K8s-XX :~$ wget -q -O -
https://raw.githubusercontent.com/k3d-io/k3d/main/install.sh | bash
```

```
Preparing to install k3d into /usr/local/bin
```

```
k3d installed into /usr/local/bin/k3d
```

```
Run 'k3d --help' to see what you can do with it.
```

- Kubectl installeren.

```
student@virt-K8s-XX :~$ sudo cat <<EOF | sudo tee
/etc/yum.repos.d/kubernetes.repo
[kubernetes]
name=Kubernetes
baseurl=https://pkgs.K8s.io/core:/stable:/v1.32/rpm/
enabled=1
gpgcheck=1
gpgkey=https://pkgs.K8s.io/core:/stable:/v1.32/rpm/repo
md.xml.key
EOF
student@virt-K8s-XX :~$ sudo dnf update
student@virt-K8s-XX :~$ sudo dnf -y install kubectl
```

- Optioneel (afgeraden): Podman configureren zodat K3D er gebruik van kan maken:
Podman in K3D is experimenteel. Daarom gaan we enkel met Docker werken voor K8S en mag je deze stap overslaan.

- Podman socket beschikbaar maken voor k3d.

```
student@virt-K8s-XX :~$ sudo systemctl enable --now
podman.socket
```

- Zorgen dat K3D naar de juiste "docker"-socket verwijst. Dit doen we door een symbolische link aan te maken.

```
student@virt-K8s-XX :~$ sudo ln -s /run/podman/podman.sock
/var/run/docker.sock
```

11.4 Cluster aanmaken

Nadat alles is geïnstalleerd, kunnen we onze eerste k3s cluster opzetten via k3d. Hiervoor kan je volgende cmd uitvoeren:

```
student@virt-K8s-XX :~$ k3d cluster create dev-cluster --servers
1 --agents 1
```

```
...
```

--servers 1 betekent dat er 1 server node wordt aangemaakt.

--agents 1 betekent dat er 1 worker node wordt aangemaakt.

Om nu te kijken welke nodes er allemaal aangemaakt zijn kan men gebruik maken volgend cmd:

NAME VERSION	STATUS	ROLES	AGE
k3d-dev-cluster-agent-0 v1.31.5+k3s1	Ready	<none>	3m5s
k3d-dev-cluster-server-0 v1.31.5+k3s1	Ready	control-plane, master	3m8s

De output van dit cmd toont duidelijk dat er 2 nodes draaien, waarvan 1 node de Control plane is en de andere node een werker.

In de praktijk zal je meestal maar één k3d-cluster tegelijk draaien.

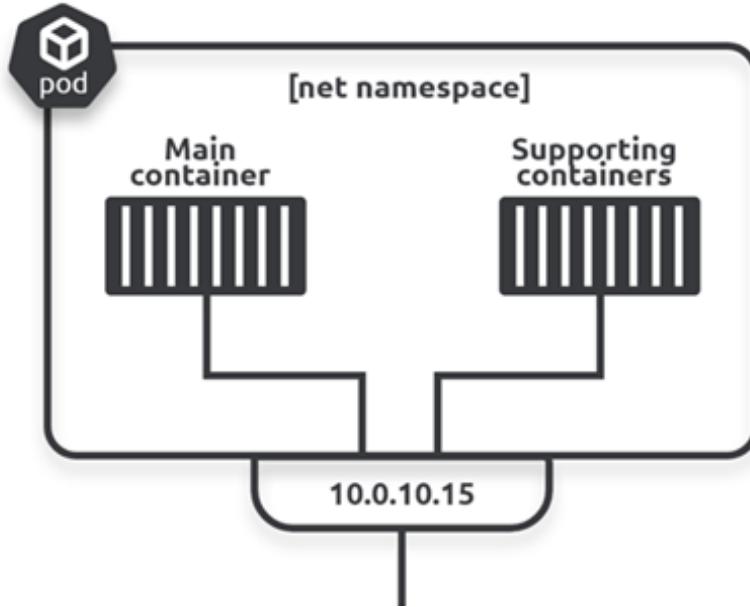
11.5 Pods

11.5.1 Inleiding

Een Kubernetes-pod is de kleinste beheersbare eenheid in Kubernetes en fungeert als een afgeschermd omgeving waarin één of meer containers draaien. De pod zelf draait geen applicatie, maar beheert de container(s) die dat doen. Containers binnen dezelfde pod delen dezelfde netwerkruimte (IP-adres) en opslag, wat zorgt voor directe en efficiënte communicatie tussen deze containers via localhost.

Dit is vergelijkbaar met Podman, waar een pod ook een verzameling containers is die nauw samenwerken. Het belangrijkste onderscheid is dat de pod de operationele context en resource-sharing regelt, terwijl de container de daadwerkelijke applicatie en bijbehorende processen draait.

Kortom: binnen Kubernetes draait elke applicatie-container in een pod, en de pod zorgt voor de isolatie, netwerk- en opslagresources die nodig zijn om die containers als één samenhangende eenheid te laten functioneren.

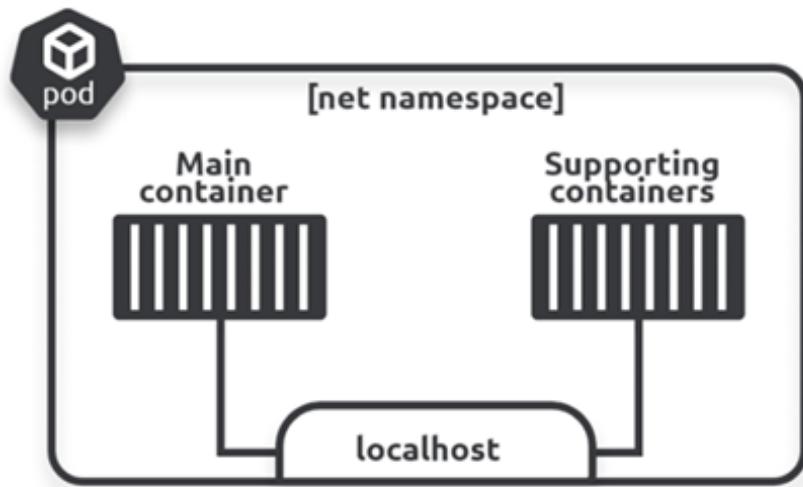


Elk Kubernetes-pod krijgt een eigen netwerkstack en kernel namespace toegewezen, wat betekent dat de pod een geïsoleerde netwerk- en systeemomgeving heeft waarin container(s) kunnen draaien. Dit maakt de pod een afgeschermd eenheid binnen het cluster.

Over het algemeen wordt aangeraden om voor elke applicatie of microservice een aparte pod te maken, zodat ze onafhankelijk kunnen schalen en beheerd worden. Het gebruik van meerdere containers in één pod is meestal voor geavanceerdere toepassingen nodig, waarbij voorbeeld een helper-container of sidecar-container (container voor extra functionaliteit) ondersteuning biedt. Dit vereist meer diepgaande kennis van Kubernetes.

Samengevat delen containers in een pod:

- De netwerkstack (IP-adres en netwerkinterfaces)
- Storage volumes (gevulde opslag)
- Kernel namespace (system-level isolatie)
- Ze draaien op dezelfde fysieke node

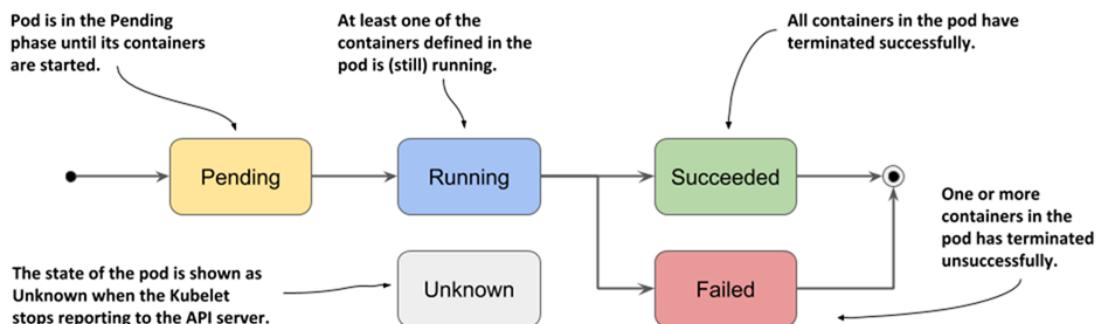


11.5.2 Lifecycle

Een pod bevindt zich steeds in een bepaalde status en heeft een eindige levenscyclus. Elke pod zal immers ooit eens stoppen met bestaan. De levenscyclus van een pod doorloopt meestal de volgende fasen:

- Pending
De pod is aangemaakt en de control plane probeert deze op een geschikte, gezonde node op te zetten, maar de containers zijn nog niet allemaal gestart.
- Running
Ten minste één container binnen de pod draait actief en de pod functioneert.
- Succeeded
Alle containers binnen de pod zijn succesvol gestopt (beëindigd), bijvoorbeeld bij batchtaken die klaar zijn.
- Failed
Eén of meer containers zijn ongepland gestopt met een fout.
- Unknown
De status van de pod kan tijdelijk niet worden bepaald.

Zolang een pod correct werkt (de container in de pod draait correct), zal deze dus in een running status blijven.

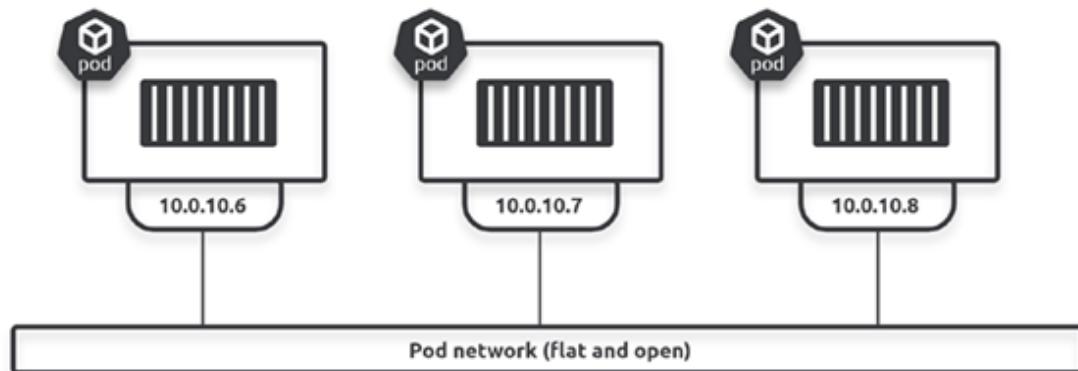


Een pod kan je niet aanpassen. Indien er iets moet gewijzigd worden aan een pod, moet je deze laten sterven en een nieuwe pod deployen.

11.5.3 Pod netwerk

In Kubernetes krijgt elke pod een eigen IP-adres toegekend, afkomstig uit een vooraf gedefinieerd IP-bereik dat wordt toegewezen aan pods in het cluster. Hierdoor heeft elke pod een uniek IP-adres dat gedeeld wordt door alle containers in die pod.

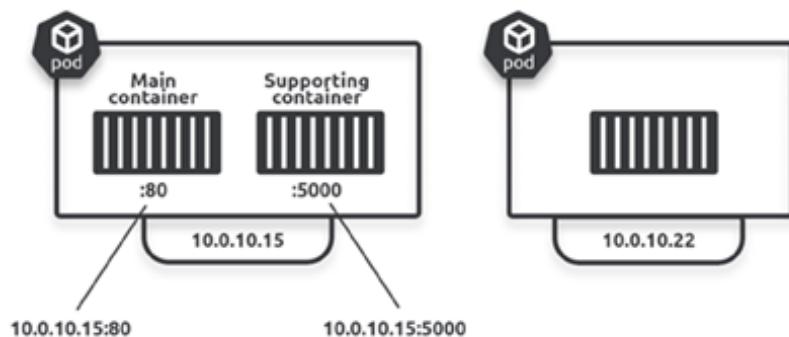
Het netwerk binnen een Kubernetes-cluster is standaard plat en open binnen de cluster, wat betekent dat alle pods zonder beperkingen met elkaar kunnen communiceren in dezelfde cluster, onafhankelijk van de node. Om deze communicatie te beperken en de veiligheid te verhogen, kan men gebruikmaken van Network Policies. Deze policies definiëren regels die bepalen welke pods met elkaar mogen communiceren, waardoor netwerkverkeer binnen het cluster kan worden afgeschermd en beveiligd.



Elke pod krijgt zijn eigen netwerk namespace, waarin alle containers binnen die pod dezelfde netwerkconfiguratie delen. Dit betekent dat alle containers in een pod hetzelfde IP-adres hebben, dat gedeeld wordt door alle containers binnen die pod. Communicatie tussen containers in dezelfde pod verloopt via localhost op de wijze van de eigen IP en hun toegewezen poorten.

Wanneer externe systemen contact willen opnemen met een container in de pod, gebeurt dat via het IP-adres van de pod zelf.

Indien men een multicontainer pod heeft, moet er dus ook op het niveau van de pod met poorten gewerkt worden om de specifieke container binnen de pod te kunnen bereiken.



11.5.4 Pod aanmaken

Een pod kan op twee manieren aangemaakt worden:

- Imperatief
 - Een pod aanmaken door het invoeren van commando's.
- Declaratief
 - Een pod (of andere objecten) aanmaken door het gebruik van een manifest bestand.

Wij gaan steeds gebruik maken van de declaratieve manier om een pod op te zetten. Hiervoor moeten we dus leren hoe een manifest bestand wordt opgebouwd.

11.5.4.1 Imperatief

Je maakt de Pod dus rechtstreeks aan door een commando in te voeren in de terminal.

```
student@serverXX:~$ kubectl run nginx-pod --image=nginx --restart=Never --port=80 --labels="zone=prod,version=v1"
```

```
pod/mypod created
```

- kubectl run: maakt een Pod aan
- nginx-pod: naam van de Pod
- --image=nginx: container image dat gebruikt wordt
- --restart=Never: pod zal niet herstarten bij crash of verwijdering
- --labels="zone=prod,version=v1": stelt labels in

Je kan de pod erna zien door onderstaande uit te voeren:

```
student@serverXX:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-pod	1/1	Running	0	4m18s

11.5.4.2 Declaratief

Een manifestbestand wordt in Kubernetes doorgaans gedefinieerd in YAML-formaat (hier: second-pod.yaml). Hieronder staat een voorbeeld van een eenvoudig manifestbestand waarmee een Pod met een Nginx-server wordt aangemaakt.

```
kind: Pod (1)
apiVersion: v1 (2)
metadata:
  name: nginx-pod2 (3)
  labels: (4)
```

```

    zone: prod
    version: v1
spec:
  containers: (5)
    - name: nginxcont (6)
      image: nginx (7)
      ports:
        - containerPort: 80 (8)

```

1. Kind
Hiermee definiëren we welk soort object we willen aanmaken.
2. apiVersion
Geeft aan welke apiversie we gebruiken in de manifest file.
3. metadata.name
Hiermee geven we een unieke naam aan het object.
4. metatada.labels
Labels worden gebruikt voor het koppelen van services aan pods (meer hierover in de paragraaf Services).
5. spec.containers
In dit deel gaan we onze specifieke containers gaan definiëren.
6. spec.containers.name
Unieke naam die we aan onze container geven.
Containernaam is niet instelbaar als je imperatief werkt. Bij imperatief maakt Kubernetes de containernaam automatisch gelijk aan de Pod-naam
7. spec.containers.image
Geeft aan welke docker image er gebruikt wordt voor de container.
8. ... containerport
Specificeert welke poort er open moet gezet worden op de container.

Zoals je kan zien zijn er heel wat belangrijke elementen die we moeten definiëren in een pod. Om deze pod te laten draaien volstaat het om de yaml file lokaal te bewaren en deze te laten deployen op de cluster.

- Maak een nieuwe cluster aan voor de declaratieve pod:

```

student@virt-k8s-XX:~$ k3d cluster create test-cluster -s 1 -a 1
INFO[0000] Prep: Network
INFO[0000] Created network 'k3d-test-cluster'
INFO[0000] Created image volume k3d-test-cluster-images
INFO[0000] Starting new tools node...
INFO[0000] Starting node 'k3d-test-cluster-tools'
INFO[0001] Creating node 'k3d-test-cluster-server-0'

```

```

INFO[0001] Creating node 'k3d-test-cluster-agent-0'
INFO[0001] Creating LoadBalancer 'k3d-test-cluster-serverlb'
INFO[0001] Using the k3d-tools node to gather environment
information
INFO[0001] HostIP: using network gateway 172.19.0.1 address
INFO[0001] Starting cluster 'test-cluster'
INFO[0001] Starting servers...
INFO[0001] Starting node 'k3d-test-cluster-server-0'
INFO[0003] Starting agents...
INFO[0004] Starting node 'k3d-test-cluster-agent-0'
INFO[0006] Starting helpers...
INFO[0006] Starting node 'k3d-test-cluster-serverlb'
INFO[0012] Injecting records for hostAliases (incl.
host.k3d.internal) and for 3 network members into CoreDNS
configmap...
INFO[0014] Cluster 'test-cluster' created successfully!
INFO[0014] You can now use it like this:
kubectl cluster-info

```

- Maak een file aan genaamd: second-pod.yaml met daarin de inhoud van de voorbeeld yaml. Let op met de spaties en dat je geen tabs gebruikt!

```

student@virt-k8s-XX:~$ nano second-pod.yaml

kind: Pod
apiVersion: v1
metadata:
  name: nginx-pod2
  labels: (4)
    zone: prod
    version: v1
spec:
  containers:
    - name: nginxcont
      image: nginx
      ports:
        - containerPort: 80

```

- Deploy de pod door gebruik te maken van volgende cmd:

```
student@virt-k8s-XX:~$ kubectl apply -f second-pod.yaml
pod/nginx-pod2 created
```

- Bekijk de status van de pod door volgende cmd:

```
student@virt-k8s-XX:~$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	...
nginx-pod2	1/1	Running	0	46s	10.42.1.6	...

Zoals je ziet zie je alleen de pods van de cluster waar je nu mee werkt.

- Je kan alle clusters opvragen met onderstaand commando.

```
student@serverXX:~$ kubectl config get-clusters
```

NAME

k3d-test-cluster

k3d-dev-cluster

- Indien je details van de pod wil weten, kan je gebruik maken van volgende cmd:

```
student@virt-k8s-XX:~$ kubectl describe pod nginx-pod2
```

Name:	nginx-pod2
Namespace:	default
Priority:	0
Service Account:	default
Node:	k3d-test-cluster-server-0/172.19.0.2
Start Time:	Fri, 07 Nov 2025 12:49:53 +0100
Labels:	version=v1 zone=prod
Annotations:	<none>
Status:	Running
IP:	10.42.1.6
...	

- Om deze pod te verwijderen, volstaat het om het volgende commando uit te voeren:

```
student@virt-k8s-XX:~$ kubectl delete pod nginx-pod2
```

```
pod "nginx-pod2" deleted
student@virt-k8s-XX:~$ kubectl get pods -o wide
No resources found in default namespace.
```

K8s maakt gebruik maakt van de Dockerhub voor het downloaden van de containers.

Natuurlijk is het aanmaken van een pod niet het enige wat we willen doen. Vaak moet men ook in interactie gaan met de pod om in de container bepaalde cmd's uit te voeren. Hiervoor kan je gebruiken maken van "kubectl exec". Dit laat toe om een cmd uit te voeren in onze pod en de output weer te geven. Indien we onze vorige pod terug uitvoeren en een ls willen uitvoeren in de container, doen we dit op de volgende manier:

```
student@virt-k8s-XX:~$ kubectl apply -f second-pod.yaml
pod/nginx-pod2 created
student@virt-k8s-XX:~$ kubectl exec nginx-pod2 -- ls -al
total 4
drwxr-xr-x.  1 root root  39 Oct 30 10:44 .
drwxr-xr-x.  1 root root  39 Oct 30 10:44 ..
lrwxrwxrwx.  1 root root   7 Aug 24 16:20 bin -> usr/bin
drwxr-xr-x.  2 root root   6 Aug 24 16:20 boot
drwxr-xr-x.  5 root root 360 Oct 30 10:44 dev
drwxr-xr-x.  1 root root  41 Oct 28 21:50 docker-entrypoint.d
-rwxr-xr-x.  1 root root 1620 Oct 28 21:49 docker-entrypoint.sh
drwxr-xr-x.  1 root root  32 Oct 30 10:44 etc
drwxr-xr-x.  2 root root   6 Aug 24 16:20 home
lrwxrwxrwx.  1 root root   7 Aug 24 16:20 lib -> usr/lib
lrwxrwxrwx.  1 root root   9 Aug 24 16:20 lib64 -> usr/lib64
drwxr-xr-x.  2 root root   6 Oct 20 00:00 media
drwxr-xr-x.  2 root root   6 Oct 20 00:00 mnt
drwxr-xr-x.  2 root root   6 Oct 20 00:00 opt
dr-xr-xr-x. 379 root root   0 Oct 30 10:44 proc
drwx-----.  2 root root  37 Oct 20 00:00 root
drwxr-xr-x.  1 root root  38 Oct 30 10:44 run
```

```
lrwxrwxrwx.    1 root root    8 Aug 24 16:20 sbin -> usr/sbin  
drwxr-xr-x.    2 root root    6 Oct 20 00:00 srv  
dr-xr-xr-x.   13 root root    0 Oct 30 08:47 sys  
drwxrwxrwt.    2 root root    6 Oct 20 00:00 tmp  
drwxr-xr-x.    1 root root   66 Oct 20 00:00 usr  
drwxr-xr-x.    1 root root   19 Oct 20 00:00 var
```

Het teken -- in het commando dient als scheidingsteken tussen de opties van kubectl exec en het commando dat je wilt uitvoeren binnen de container van de pod. Alles wat na -- komt, wordt gezien als het commando dat binnen de container moet worden uitgevoerd.

We kunnen het “kubectl exec” ook gebruiken voor een interactieve sessie met de pod op te starten:

```
student@virt-k8s-XX:~$ kubectl exec -it nginx-pod2 -- bash  
root@nginx-pod2:/# apt update  
Hit:1 http://deb.debian.org/debian trixie InRelease  
Hit:2 http://deb.debian.org/debian trixie-updates InRelease  
Hit:3 http://deb.debian.org/debian-security trixie-security InRelease  
  
Reading package lists... Done  
root@hello-pod:/# apt upgrade  
Reading package lists... Done  
Building dependency tree... Done  
Reading state information... Done  
Calculating upgrade... Done  
  
0 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.  
root@nginx-pod2:/# exit  
exit  
student@virt-k8s-XX:~$
```

11.6 Kubectl

11.6.1 Algemeen

Bij het aanmaken van onze eerste pods, hebben we reeds een paar keer leren werken met het kubectl cmd. Dit is het standaard tool voor het werken met een K8s cluster.

Deze tool heeft enorm veel mogelijkheden, dewelke opgebouwd zijn volgens een duidelijke syntax.

kubectl [command] [TYPE] [NAME] [flags]

- command
Commando of operatie, bv apply, create, delete, get
- type
Het type van de resource of object
- name
De naam van de resource of object
- flags
Optionele vlaggen

Voorbeelden:

- kubectl create -f mypod.yaml
- kubectl get pods
- kubectl get pod mypod
- kubectl delete pod mypod

De installatie van de cluster gebeurt in ons geval met k3d, terwijl het werken met de functionaliteit van de cluster via kubectl verloopt.

11.6.2 Context

Bij het onderdeel over het opzetten van clusters zagen we dat het mogelijk is om meerdere Kubernetes-clusters op één server te draaien.

Met kubectl kun je met al deze clusters werken via zogenoemde contexten.

Een context (werkomgeving) is een configuratie-item in de kubeconfig die een specifieke combinatie beschrijft van:

- een cluster (met zijn API-server),
- een gebruiker (voor authenticatie (authenticatie wordt verder niet besproken)),
- en een namespace (dit bespreken we later).

Door een context te gebruiken weet kubectl precies:

met welk cluster, namens welke gebruiker en binnen welke namespace het moet communiceren.

Zo kun je eenvoudig wisselen tussen verschillende omgevingen (bijvoorbeeld de dev, test en prod-omgeving) zonder telkens je configuratie te moeten aanpassen.

We laten de volledige cubeconfig zien.

```
student@serverXX:~$ kubectl config view
```

```

apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://0.0.0.0:45273
    name: k3d-dev-cluster
- cluster:
    certificate-authority-data: DATA+OMITTED
    server: https://0.0.0.0:44735
    name: k3d-test-cluster
contexts:
- context:
    cluster: k3d-dev-cluster (1)
    user: admin@k3d-dev-cluster (2)
    name: k3d-dev-cluster (3)
- context:
    cluster: k3d-test-cluster
    user: admin@k3d-test-cluster
    name: k3d-test-cluster
current-context: k3d-test-cluster
kind: Config
preferences: {}
users:
- name: admin@k3d-dev-cluster
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED
- name: admin@k3d-test-cluster
  user:
    client-certificate-data: DATA+OMITTED
    client-key-data: DATA+OMITTED

```

1. cluster: k3d-dev-cluster
Dit geeft aan met welk cluster deze context verbinding maakt.
Het verwijst naar één van de clusters die gedefinieerd zijn onder clusters.
In jouw geval: het cluster dat draait op <https://0.0.0.0:44909>.
2. user: admin@k3d-dev-cluster
Dit geeft aan welke gebruiker (authenticatie) gebruikt wordt om toegang te krijgen tot het cluster.
Het verwijst naar een entry in users: in de kubeconfig.
We hebben dit niet ingesteld.
3. name: k3d-dev-cluster
Dit is de naam van de context zelf.
Een context is een combinatie van cluster + user (+ namespace optioneel).
Je gebruikt deze naam om te wisselen van context

Toont de naam van de actieve context.

```
student@serverXX:~$ kubectl config current-context
k3d-test-cluster
```

We tonen nu een overzicht van alle contexts.

```
student@serverXX:~$ kubectl config get-contexts
CURRENT   NAME          CLUSTER          AUTHINFO ...
           k3d-dev-cluster   k3d-dev-cluster   admin@k3d-dev-cluster ...
*         k3d-test-cluster   k3d-test-cluster   admin@k3d-test-cluster
```

Zoals verwacht zie je dat k3d-test-cluster nu actief is.

We switchen nu naar de context k3d-dev-cluster.

```
student@serverXX:~$ kubectl config use-context k3d-dev-cluster
Switched to context "k3d-dev-cluster".
```

Als je nu de pods opvraagt krijg je onderstaande.

```
student@serverXX:~$ kubectl get pods
NAME        READY   STATUS    RESTARTS   AGE
nginx-pod   1/1     Running   0          64m
```

We willen nu bijvoorbeeld een nieuwe context maken met de naam dev-cluster, met gebruiker admin@k3d-dev-cluster en namespace default.

```
student@serverXX:~$ kubectl config set-context dev-context \
--cluster=k3d-dev-cluster --user=admin@k3d-dev-cluster \
--namespace=default
```

We tonen nu terug een overzicht van alle contexts.

```
student@serverXX:~$ kubectl config get-contexts
CURRENT   NAME          CLUSTER          AUTHINFO ...
           dev-context    k3d-dev-cluster   admin@k3d-dev-cluster ...
*         k3d-dev-cluster   k3d-dev-cluster   admin@k3d-dev-cluster
           k3d-test-cluster   k3d-test-cluster   admin@k3d-test-cluster
```

We switchen terug naar de context k3d-test-cluster.

```
student@serverXX:~$ kubectl config use-context k3d-test-cluster
Switched to context "k3d-first-pod".
```

11.6.3 CRUD

Kubectl maakt het mogelijk om de vier basisbewerkingen binnen een Kubernetes-cluster uit te voeren volgens het CRUD-principe:

- Create (aanmaken)
- Read (uitlezen)
- Update (bijwerken)
- Delete (verwijderen)

**CRUD-
actie**

Betekenis

Voorbeeld commando's

Create	Maak een resource aan	kubectl apply -f <naam>.yaml
		kubectl create -f <naam>.yaml
Read	Bekijk bestaande resources	kubectl get pods
		kubectl describe svc my-service
Update	Pas een bestaande resource aan	kubectl edit deployment my-app
		kubectl apply -f updated.yaml
Delete	Verwijder een resource	kubectl delete pod my-pod
		kubectl delete -f deployment.yaml

11.6.3.1 Create

Voor het aanmaken van een object binnen K8s kan men gebruik maken van 2 commando's:

- kubectl create

```
kubectl create <type> <parameters>
kubectl create -f <path to manifest>
```

Dit commando zal een resource aanmaken op een imperatieve wijze (via de stdin) of door het aanleveren van een manifest file door gebruik te maken van de -f parameter.

- kubectl apply

```
kubectl apply -f <path to manifest>
```

Het commando kubectl apply -f <path naar manifest> maakt, net als het create-commando, een resource aan op basis van de opgegeven manifestfile. Het verschil is dat apply ook wordt gebruikt om wijzigingen aan een bestaande resource door te voeren, waarbij het de configuratie bijwerkt zonder de hele resource te verwijderen en opnieuw aan te maken.

11.6.3.2 Read

Kubectl laat ook toe om details op te vragen van de verschillende resources die men heeft aangemaakt.

```
kubectl get <type>
kubectl get <type> <name>
kubectl get <type> <name> -o <output format>
```

Voorbeeld:

```
student@virt-k8s-XX:~$ kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
nginx-pod2  1/1     Running   0          22h

student@virt-k8s-XX:~$ kubectl get pod -o wide
NAME      READY   STATUS    RESTARTS   AGE   IP           ...
nginx-pod2  1/1     Running   0          22h   10.42.1.7   ...
```

11.6.3.3 Update

Met kubectl edit <type> <object name> kunnen we resources aanpassen. Dit commando zal een kopie van het object openen in een teksteditor om aan te passen.

Voorbeeld:

```
kubectl edit pod nginx-pod2 #voer best niet uit/verlaten: typ :q
```

Omdat handmatige wijzigingen meestal niet worden vastgelegd in versiebeheer of manifestbestanden, wordt het gebruik van kubectl edit in productieomgevingen meestal afgeraden. Het kan leiden tot configuratieafwijkingen die lastig te reproduceren of traceren zijn.

11.6.3.4 Delete

De laatste CRUD bewerking die we hebben, is het verwijderen van resources. Dit doen we aan de hand van het volgende commando:

```
kubectl delete <type> <name>
```

Voorbeeld:

```
student@virt-k8s-XX:~$ kubectl get pod
NAME      READY   STATUS    RESTARTS   AGE
nginx-pod2  1/1     Running   0          4s

student@virt-k8s-XX:~$ kubectl delete pod nginx-pod2
pod "nginx-pod2" deleted

student@virt-k8s-XX:~$ kubectl get pod
No resources found in default namespace.
```

11.6.4 Kubectl cheat sheet

<https://Kubernetes.io/docs/reference/kubectl/cheatsheet/>

11.7 Namespaces

11.7.1 Basis

In Kubernetes dienen namespaces om resources logisch te groeperen en te isoleren binnen één cluster. Ze zijn als het ware virtuele omgevingen binnen hetzelfde fysieke (of virtuele) cluster.

Wanneer je een nieuw cluster opstelt, zijn er standaard vier namespaces aanwezig:

Namespace	Doel
default	De standaard namespace. Als je geen -n opgeeft bij een kubectl-commando, gebruikt Kubernetes automatisch deze namespace. Alle "gewone" resources komen hier terecht, tenzij je anders specificeert.
kube-system	Hier draait de infrastructuur van Kubernetes zelf, zoals de API-server, kube-dns (CoreDNS), scheduler, controller-manager, enz. Je hoort hier meestal niet zelf applicaties te plaatsen.
kube-public	Bevat publieke informatie, zoals cluster-informatie, die voor iedereen toegankelijk is (wordt zelden gebruikt).
kube-node-lease	Wordt intern door Kubernetes gebruikt om node leases bij te houden. Dit helpt de control plane snel te detecteren of een node nog leeft. (Elke node krijgt hier een "lease object").

```
student@virt-k8s-XX:~$ kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	17m
kube-node-lease	Active	17m
kube-public	Active	17m
kube-system	Active	17m

Namespaces maken het mogelijk om je Kubernetes-cluster logisch te partitioneren. Dit helpt je om:

- Meerdere teams veilig in één cluster te laten werken zonder elkaar's resources te beïnvloeden.

- Verschillende omgevingen zoals ontwikkeling (dev), testen (test) en productie (prod) te scheiden.
- Resource quota's en beleidsregels (policies) af te dwingen.
- Het beheer van je cluster overzichtelijk en beheersbaar te houden.

Zo krijg je betere isolatie, toegangscontrole en resourcebeheer binnen één shared clusteromgeving.

Een nieuwe namespace aanmaken doe je door gebruik te maken van het kubectl create ns <name>

Voorbeeld:

```
student@virt-k8s-XX:~$ kubectl create ns pzl
namespace/pxl created
```

```
student@virt-k8s-XX:~$ kubectl get namespaces
```

NAME	STATUS	AGE
default	Active	22m
kube-node-lease	Active	22m
kube-public	Active	22m
kube-system	Active	22m
pxl	Active	27s

Uiteraard kan je ook een namespace aanmaken op de declaratieve wijze. Hiervoor volstaat het om volgende YAML (ns.yaml) file aan te maken:

```
apiVersion: v1
kind: Namespace
metadata:
  name: dev
```

Vervolgens moeten we deze nog aanmaken:

```
student@virt-k8s-XX:~$ kubectl apply -f ns.yaml
namespace/dev created
```

```
student@virt-k8s-XX:~$ kubectl get namespaces
NAME      STATUS   AGE
default   Active   25h
dev       Active   16s
```

kube-node-lease	Active	25h
kube-public	Active	25h
kube-system	Active	25h
pxl	Active	5m13s

11.7.2 Resources toewijzen aan NS

By default zal elke pod (of elke andere resource) die je aanmaakt in de default namespace worden geplaatst. Indien je met verschillende namespaces werkt, is dit niet meteen het gewenste resultaat. Je wilt immers je pods (of andere resources) in specifieke namespaces plaatsen. Afhankelijk van welke wijze je pods (of andere resources) aanmaakt, is hiervoor een andere werkwijze nodig:

- Imperatief
Indien je bijvoorbeeld een pod wilt aanmaken in de dev namespace moet je dit aan de hand van de -n parameter.

```
student@virt-k8s-XX:~$ kubectl run nginx-dev --image nginx -n dev
pod/nginx-dev created
```

- Declaratief
Bij het aanmaken van de manifest file volstaat het om bij de metadata de correcte namespace te specificeren.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  namespace: dev
spec:
  containers:
  - name: nginx
    image: nginx
```

11.7.3 Resources opvragen in NS

Ook bij het opvragen van de details van de resource moeten we steeds de -n parameter meegeven. Kubectl zal immers steeds in de default namespace zoeken naar de resources die hij wil opvragen.

```
student@virt-k8s-XX:~$ kubectl get pods
kubectl run nginx-dev --image nginx -n dev
```

```
student@virt-k8s-XX:~$ kubectl get pods -n dev
NAME      READY   STATUS    RESTARTS   AGE
nginx-dev  1/1     Running   0          4m39s
```

Indien je met meerdere namespaces aan het werken bent, kan het gebruik van de -n parameter snel lastig worden om mee te werken. Zeker indien je verschillende commando's wilt uitvoeren voor een

bepaalde namespace. Je kan echter je kubeconfig aanpassen zodat deze standaard in een bepaalde namespace werkt. Dit doe je aan de hand van het volgende cmd:

```
kubectl config set-context --current --namespace <ns-name>
```

Voorbeeld:

```
student@virt-k8s-XX:~$ kubectl config set-context --current --namespace dev  
Context "k3d-test-nginx" modified.
```

```
student@virt-k8s-XX:~$ kubectl get pods  
NAME        READY   STATUS    RESTARTS   AGE  
nginx-dev   1/1     Running   0          9m18s
```

Zorg wel dat je steeds goed beseft in welke namespace je aan het werken bent.

Je kan ook alle pods in alle namespaces als volgt opvragen.

```
student@virt-k8s-XX:~$ kubectl get pods --all-namespaces  
...
```

11.8 Services

11.8.1 Algemeen

Hoewel het netwerk binnen een Kubernetes-cluster standaard plat en open is, waardoor alle pods rechtstreeks met elkaar kunnen communiceren in een cluster, is het niet aan te raden om pods direct via IP-adressen te laten communiceren.

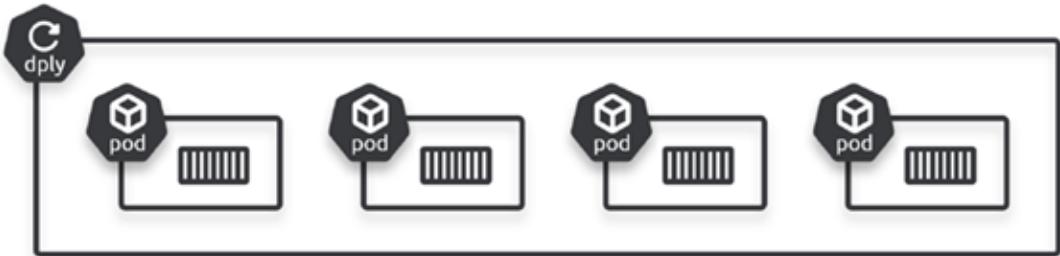
Stel dat je een pod hebt waarop een website draait en een andere pod waarop een database draait. In de configuratie van de website kun je het IP-adres van de database-pod gebruiken om verbinding te maken. Dit werkt, omdat pods in een Kubernetes-cluster standaard onbeperkt met elkaar kunnen communiceren.

Echter, als je de database-pod vervolgens verwijdert en opnieuw aanmaakt — wat gebruikelijk is bij updates of scaling — dan krijgt de nieuwe pod een ander IP-adres. Hierdoor verliest je website de verbinding, omdat het nog steeds naar het oude IP verwijst.

Daarnaast, als je meerdere website-pods wil draaien voor schaalbaarheid, moet je een manier hebben om die pods collectief te benaderen, bijvoorbeeld door een enkele toegangspoort voor externe gebruikers te creëren.



↖_(ၷ)ၷ

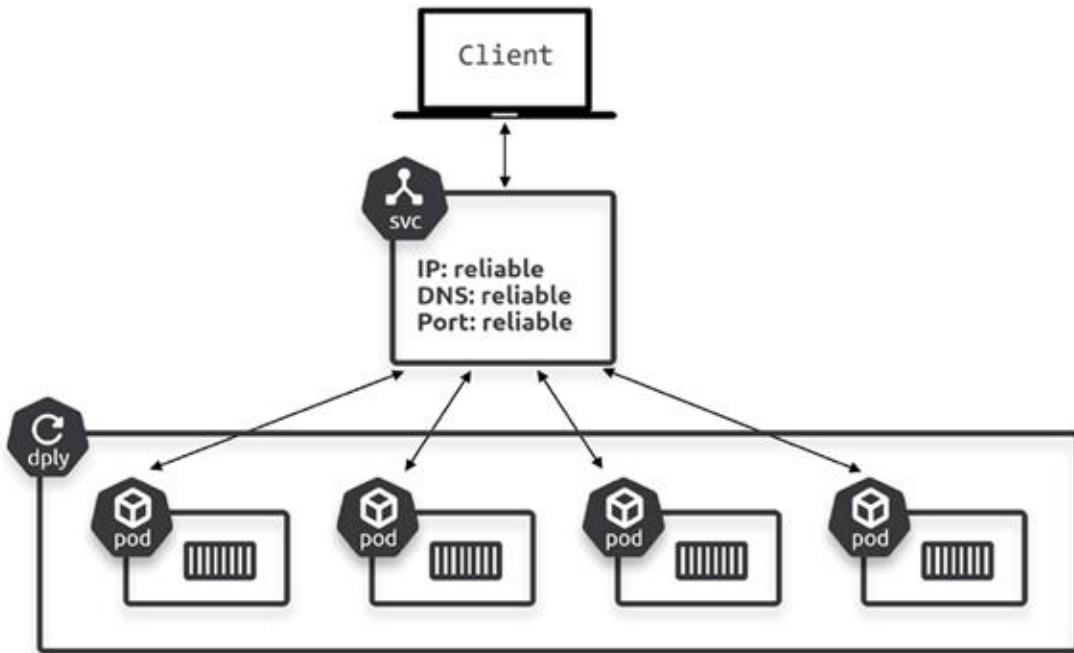


Om de problemen uit de vorige voorbeelden op te lossen, maak je binnen Kubernetes gebruik van Services. Services zorgen ervoor dat pods binnen het cluster altijd op een consistente en betrouwbare manier bereikbaar zijn, zelfs als de pods dynamisch veranderen, bijvoorbeeld door opschalen of herstarten.

Een Service is een speciaal type resource dat fungeert als een stabiel aanspreekpunt voor een groep pods. Dit maakt het mogelijk om pods te bereiken via een onveranderlijk netwerkendpoint, onafhankelijk van de levenscyclus van de individuele pods.

Elke Service heeft een aantal vaste eigenschappen die gedurende de hele levensduur van de Service blijven bestaan:

- Een vast (clusterbreed) IP-adres
- Een consistente DNS-naam binnen het cluster
- Een of meerdere TCP- of UDP-poorten waarlangs de service bereikbaar is



Door gebruik te maken van een Service in Kubernetes beschik je over een betrouwbaar en stabiel netwerkendpoint dat toegang biedt tot een groep pods, ongeacht veranderingen in het aantal pods achter deze Service. Dit betekent dat het aantal pods naadloos kan schalen – omhoog of omlaag – zonder dat gebruikers of verbonden applicaties iets hoeven te veranderen.

Een kracht van een Service is dat het automatisch zorgt voor load balancing. Alle pods die gekoppeld zijn aan een Service via labels worden continu gemonitord. Als een pod niet meer gezond is, zal de Service deze automatisch uitsluiten van de load balancing, zodat alleen gezonde pods verkeer ontvangen.

Kubernetes kent verschillende servicetypes met elk hun eigen kenmerken en gebruik:

Type	Toegang vanaf	Gebruik
ClusterIP	Alleen binnen het cluster	Standaardtype, voor interne communicatie tussen pods en services
NodePort	Buiten het cluster via node-IP en poort	Eenvoudige externe toegang, vaak voor lokale of testomgevingen
LoadBalancer	Buiten het cluster via extern IP	Cloud-native toegang met geïntegreerde load balancer (bv. AWS, GCP, Azure)

Hoewel het Kubernetes-netwerk standaard open is, is het altijd aan te raden om een Service te definiëren wanneer een Pod via specifieke poorten bereikbaar moet zijn

11.8.2 Endpoint slices

Zoals eerder besproken voert een Service in Kubernetes verkeer-distributie (load balancing) uit over de achterliggende pods. Hiervoor heeft de Service een actuele lijst nodig van actieve en gezonde endpoints (pods) die bij de Service horen.

Deze lijst wordt samengesteld op basis van labelselectie en vastgelegd in EndpointSlices.

Wanneer een Service wordt aangemaakt, creëert Kubernetes automatisch één of meer bijbehorende EndpointSlices. Deze bevatten subsets van de IP-adressen en poorten van de pods die via de labelselector aan de Service gekoppeld zijn.

De control plane (met name de kube-controller-manager en endpoint controller) houdt continu bij welke pods aan de labelselector voldoen. Nieuwe pods worden aan de EndpointSlices toegevoegd zodra ze beschikbaar zijn. Verwijderde of niet-gezonde pods worden eruit verwijderd.

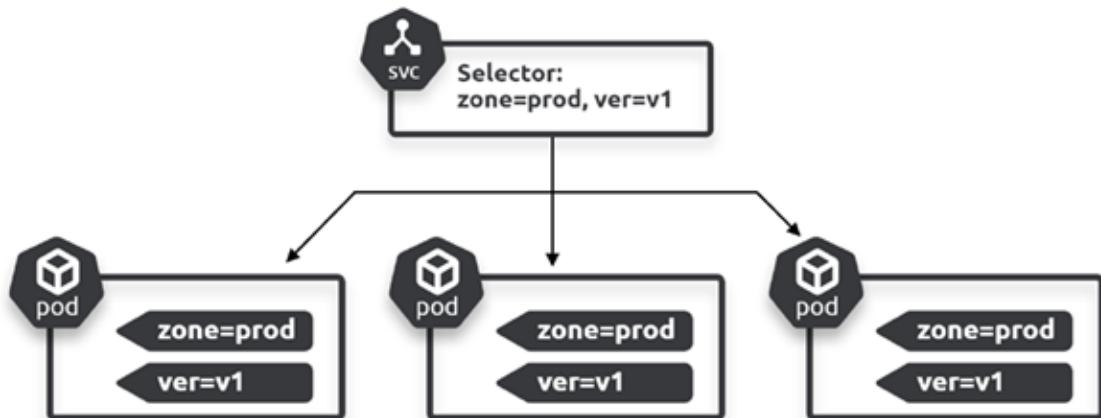
Op deze manier zorgt Kubernetes ervoor dat de lijst met actieve en gezonde pods altijd up-to-date is — essentieel voor betrouwbare load balancing.

Let op: alleen pods die Ready zijn, worden in de EndpointSlices opgenomen.

11.8.3 Services aanmaken

Het aanmaken van een Service kan op twee manieren, maar wij richten ons hier uitsluitend op de declaratieve aanpak met manifestbestanden.

Voordat we ingaan op het maken van een Service, is het belangrijk te begrijpen hoe een Service wordt gekoppeld aan pods. In Kubernetes koppelen we altijd pods aan een Service, nooit andersom. Deze koppeling gebeurt via labels.



Zolang de manifestfile van een pod de juiste labels bevat die overeenkomen met de labelselector van een Service, zal de pod automatisch worden geregistreerd bij die Service en opgenomen worden in de load balancing.

Bovenaan zie je een Service (svc) met een labelselector: zone=prod, ver=v1.

Onder de Service staan drie pods, elk voorzien van dezelfde labels: zone=prod en ver=v1.

Hiermee wordt duidelijk dat het labelmatchen tussen pods en services de sleutel is tot automatische koppeling en load balancing binnen Kubernetes.

11.8.3.1 Cluster IP

Zoals aangehaald in het overzicht van soorten services zorgt dit soort van service enkel voor communicatie in een cluster tussen pods en andere services. Volgende manifestfile svc.yaml zorgt voor het aanmaken van een ClusterIP-service.

```
apiVersion: v1
kind: Service (1)
metadata:
  name: nginx-svc (2)
spec:
  type: ClusterIP (3)
  selector:
    app: nginx (4)
  ports:
    - port: 8080 (5)
      targetPort: 80 (6)
      protocol: TCP
```

1. kind: Service

Dit geeft aan wat voor soort Kubernetes-object je aanmaakt.

In dit geval is het een Service

2. name: nginx-svc

De naam van de Service.

Deze naam wordt gebruikt:

- om de Service in de cluster te identificeren;
- en als DNS-naam binnen de cluster.

3. type: ClusterIP

Bepaalt hoe de Service toegankelijk is binnen of buiten de cluster.

ClusterIP → enkel binnen de cluster bereikbaar (standaard).

4. App: nginx

Dit koppelt de Service aan de juiste Pods. Dit betekent dat alle Pods die het label app=nginx hebben, behoren tot deze Service.

5. port: 8080

De poort waarop de Service binnen de cluster luistert.

Andere Pods binnen dezelfde namespace kunnen dus verbinden naar nginx-svc:8080.

6. targetPort: 80

De poort in de container (of Pod) waarop de applicatie effectief luistert.

De Service stuurt inkomend verkeer op port: 8080 door naar targetPort: 80 in de Pods.
client → nginx-svc:8080 → (doorsturen) → nginx-pod:80

Vervolgens moeten we ook nog een pod aanmaken met de effectieve server (svc-nginx.yaml).

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-pod
  labels:
    app: nginx
spec:
  containers:
    - name: nginx
      image: nginx
      ports:
        - containerPort: 80
```

Daarnaast maken we ook nog een gewone debian pod aan dewelke we gaan gebruiken als client (svc-debian.yaml).

```
apiVersion: v1
kind: Pod
metadata:
  name: debian-pod
spec:
  containers:
    - name: debian
      image: debian:latest
      command: ["sleep", "infinity"]
      tty: true
```

The image shows three terminal windows side-by-side:

- Top Window:** Contains the `svc.yaml` file. It defines a Service named `nginx-svc` with type `ClusterIP`. The selector is `app: nginx`. A port mapping is shown where `port: 8080` maps to `targetPort: 80` via `TCP`.
- Middle Window:** Contains the `svc-nginx.yaml` file. It defines a Deployment named `nginx-pod` with one container named `nginx` running `nginx` image. The port `containerPort: 80` is specified.
- Bottom Window:** Contains the `svc-deb.yaml` file. It defines a Pod named `debian-pod` with one container named `debian` running `debian:latest` image. The command is set to `["sleep", "infinity"]` and `tty: true`.

Nu moeten we onze deployments ook daadwerkelijk uitvoeren. Het is aan te raden om eerst de Service te deployen en daarna de Pods.

```
student@virt-k8s-XX:~$ kubectl apply -f svc.yaml
```

```
service/nginx-svc created
```

```
student@virt-k8s-XX:~$ kubectl get svc -o wide
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE	SELECTOR
nginx-svc	ClusterIP	10.43.126.13	<none>	8080/TCP	73s	app=nginx

```
student@virt-k8s-XX:~$ kubectl apply -f svc-deb.yaml
```

```
pod/debian-pod created
```

```
student@virt-k8s-XX:~$ kubectl apply -f svc-nginx.yaml
```

```
pod/nginx-pod created
```

```
student@virt-k8s-XX:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
debian-pod	1/1	Running	0	33s

```
nginx-pod     1/1     Running     0          5m37s
```

We kunnen nu de Debian pod gebruiken om onze service te testen.

```
student@virt-k8s-XX:~$ kubectl exec -it debian-pod -- sh  
  
# apt update && apt install -y curl  
# curl http://nginx-svc:8080  
  
<!DOCTYPE html>  
  
<html>  
  
<head>  
  
<title>Welcome to nginx!</title>  
  
<style>  
  
...  

```

Uit de bovenstaande output blijkt duidelijk dat de Service (nginx-svc) correct functioneert. De Debian-pod kan via de interne DNS-naam nginx-svc:8080 succesvol verbinding maken met de Nginx-pod, wat bevestigt dat de Service goed werkt en het verkeer correct doorstuurt.

11.8.3.2 NodePort

11.8.3.2.1 Basis

Waar een ClusterIP enkel binnen de cluster een vast aanspreekpunt biedt voor interne communicatie, gaat een NodePort een stap verder.

Met een NodePort maak je de service namelijk ook extern bereikbaar — via een specifieke poort op elke node in de cluster.

In het onderstaande voorbeeld zie je hoe je een NodePort-service kunt aanmaken met behulp van een manifestbestand (nodeport.yaml).

```
apiVersion: v1  
  
kind: Service (1)  
  
metadata:  
  
  name: nginx-srv2 (2)  
  
spec:
```

```

type: NodePort (3)

selector:

app: nginx (4)

ports:

  - port: 8080 (5)

    targetPort: 80 (6)

    nodePort: 30050 (7)

```

1. kind: Service

Dit geeft aan wat voor soort Kubernetes-object je aanmaakt.

In dit geval is het een Service

2. name: nginx-srv2

De naam van de Service.

Deze naam wordt gebruikt:

- om de Service in de cluster te identificeren;
- en als DNS-naam binnen de cluster.

3. type: NodePort

Bepaalt hoe de Service toegankelijk is binnen of buiten de cluster.

NodePort → maakt de Service bereikbaar via een vaste poort op elke node, ook van buiten de cluster.

4. App: nginx

Dit koppelt de Service aan de juiste Pods. Dit betekent dat alle Pods die het label app=nginx hebben, behoren tot deze Service.

5. port: 8080

Dit is de poort waarop de Service luistert binnen de cluster.

Andere Pods (of Services) kunnen via demo-node:8080 verbinding maken.

6. targetPort: 80

De poort in de container (of Pod) waarop de applicatie effectief luistert.

De Service stuurt inkomend verkeer op port: 8080 door naar targetPort: 80 in de Pods.

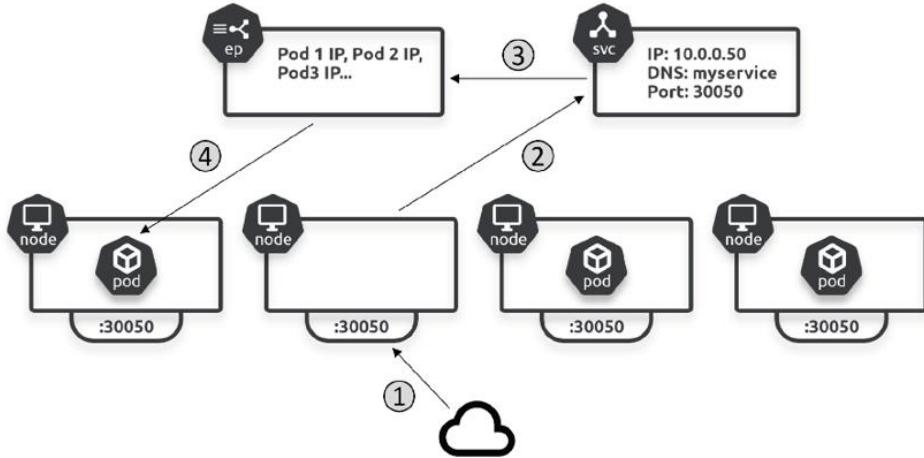
7. nodePort: 30050

Dit is de poort op elke node van de cluster waarmee de Service van buiten de cluster bereikbaar is.

Externe clients kunnen verbinding maken via <NodeIP>:30050, waarna het verkeer automatisch wordt doorgestuurd naar de Service (port: 8080) en vervolgens naar de bijbehorende Pod (targetPort: 30050).

Als je nodePort niet expliciet opgeeft, kiest Kubernetes automatisch een poort in het bereik 30000–32767. Je mag zelf ook geen poorten buiten dat bereik kiezen voor de nodePort.

Door zelf een nodePort te definiëren, weet je precies welke poort extern bereikbaar is, wat handig is voor firewalls of voorspelbare toegang.



Wanneer je deze NodePort Service deployt, wordt op elke node in het cluster poort 30050 geopend. Via deze poort kunnen externe clients het cluster binnengaan. De communicatie verloopt volgens de volgende stappen, zoals in de afbeelding:

- Een externe client maakt verbinding met het IP-adres van een willekeurige node op poort 30050.
- Het inkomende verkeer wordt ontvangen door de NodePort Service op die node.
- De Service raadpleegt de bijbehorende EndpointSlice, waar een actuele lijst van gezonde pods (endpoints) voor deze Service wordt bijgehouden.
- Op basis van load balancing stuurt de Service de aanvraag door naar een van de gezonde pods (1 in tekening), ongeacht op welke node deze pod draait.

Belangrijk: het maakt niet uit op welke node je cluster binnengaat; het verkeer wordt altijd op dezelfde manier verwerkt en gespreid over de beschikbare, gezonde pods. Welke pod de aanvraag uiteindelijk afhandelt, wordt bepaald door de loadbalancing van de Service.

Je mag het IP-adres van eender welke worker node gebruiken omdat NodePort-services in Kubernetes op alle nodes van het cluster dezelfde externe poort openen.

Nu moeten we onze deployments ook daadwerkelijk uitvoeren.

```
student@virt-k8s-XX:~$ kubectl apply -f nodeport.yaml
service/nginx-srv2 created
```

Bij k8s zou onderstaande nu werken.

```
student@serverXX:~$ curl localhost:30050
curl: (7) Failed to connect to localhost port 30050 after 0 ms:
Could not connect to server
```

11.8.3.2 K3d en nodeports

De fout treedt op omdat we met k3d werken... Bij een echte K8s installatie zou het dus werken.

Kubernetes zelf draait in Docker-containers bij k3d. Je kan dat eenvoudig checken.

```
student@serverXX:~$ docker ps --filter name=k3d
```

CONTAINER ID	IMAGE	COMMAND
207bbb1703fa	ghcr.io/k3d-io/k3d-proxy:5.8.3	"/bin/sh -c nginx-pr..." ...
3964528f5a9e	rancher/k3s:v1.31.5-k3s1	"/bin/k3d-entrypoint..." ...
ce4898349338	rancher/k3s:v1.31.5-k3s1	"/bin/k3d-entrypoint..." ...
5173339da660	ghcr.io/k3d-io/k3d-proxy:5.8.3	"/bin/sh -c nginx-pr..." ...
c2d2fac65183	rancher/k3s:v1.31.5-k3s1	"/bin/k3d-entrypoint..." ...
4d3afdc4107f	rancher/k3s:v1.31.5-k3s1	"/bin/k3d-entrypoint..." ...

Daardoor zit de NodePort-routing binnen dat containernetwerk, en is poort 30050 niet automatisch beschikbaar op je VM-host tenzij je die poort expliciet hebt gepubliceerd.

Je kan dat gelukkig (op 2 manieren) oplossen...

- Optie A

Voor kubectl port-forward op de service zoals hieronder staat om te forwarden.

```
student@serverXX:~$ kubectl port-forward svc/nginx-srv2
30050:8080 -n dev &

[1] 44857

student@serverXX:~$ Forwarding from 127.0.0.1:30050 -> 80
Forwarding from [::1]:30050 -> 80
```

```
student@serverXX:~$ curl localhost:30050

Handling connection for 30050

<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
...
```

Het werkt 😊.

- Optie B

Als je persistent NodePort-toegang wil vanaf de VM (zonder port-forward na reboot), maak (of hermaak) je de cluster waarbij je de po(o)rt(en) publiceert op de loadbalancer.

We zullen hiervoor eerst de bestaande clusters verwijderen (om niet te veel resources te gebruiken).

```
student@serverXX:~$ k3d cluster list

NAME          SERVERS   AGENTS   LOADBALANCER
dev-cluster    1/1       1/1      true
first-pod      1/1       1/1      true

student@serverXX:~$ k3d cluster delete dev-cluster
...
student@serverXX:~$ k3d cluster delete test-cluster
...
```

We maken nu een nieuwe cluster aan met publicatie van poort 30050 via de k3d loadbalancer.

```
student@serverXX:~$ k3d cluster create mycluster -p
30050:30050@server:0
...
...
```

We voeren nu de nodige deployments uit.

```
student@serverXX:~$ kubectl apply -f nodeport.yaml
service/nginx-srv2 created
student@serverXX:~$ kubectl apply -f svc-nginx.yaml
pod/nginx-pod created
```

We trachten nu de webpagina op te vragen.

```
student@serverXX:~$ curl localhost:30050
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
...
```

Ook dit werkt 😊.

11.8.3.3 Loadbalancer

LoadBalancer Services bieden eenvoudige externe toegang tot je Kubernetes workloads door automatisch een internet-facing load balancer in te stellen via je cloudprovider. Je krijgt een publiek IP-adres of DNS-naam die altijd bereikbaar is en die automatisch verkeer verdeelt naar je Service. Hierdoor hoef je geen IP-adressen van individuele nodes te beheren en kun je eventueel een eigen, herkenbare DNS-naam koppelen voor gemakkelijke toegang.

Let op: LoadBalancer Services zijn alleen beschikbaar op cloudplatforms die dit ondersteunen, zoals AWS, Azure of GCP.

In deze cursus focussen we ons alleen op ClusterIP en NodePort, omdat een LoadBalancer Service in een lokale testomgeving meestal niet beschikbaar is.

11.9 Deployments

11.9.1 Algemeen

Een Deployment is een Kubernetes-resource waarmee je eenvoudig applicaties uitrolt, bijwerkt en beheert. Je definieert hierin hoeveel pods je wilt draaien, en Kubernetes zorgt ervoor dat dit aantal altijd up-to-date blijft (self-healing). Bij wijzigingen aan je applicatie voert een Deployment gecontroleerde updates uit (rolling updates) en maakt het eventueel terugdraaien (rollback) mogelijk. Elke microservice krijgt idealiter een eigen Deployment, zodat je elke service onafhankelijk kunt updaten en schalen voor maximale flexibiliteit.

11.9.2 ReplicaSets

Deployments gebruiken ReplicaSets om het gewenste aantal pods te beheren. ReplicaSets zorgen ervoor dat altijd het juiste aantal werkende pods actief is: als er één uitvalt, wordt er automatisch een nieuwe gestart. Ze maken zo self-healing en eenvoudig schalen mogelijk.

11.9.3 Voorbeeld

11.9.3.1 Basis

Een deployment resource in Kubernetes maak je aan met een YAML-manifestbestand. Dit manifest beschrijft precies hoe jouw applicatie uitgerold moet worden: hoeveel pods je wilt, welke container-image er gebruikt wordt, en op welke poort je app bereikbaar is.

```
apiVersion: apps/v1  
kind: Deployment (1)  
  
metadata:  
  name: nginx-deployment (2)  
  
spec:
```

```

replicas: 2 (3)

selector:

  matchLabels:

    app: nginx

template: (4)

metadata:

  labels:

    app: nginx

spec:

  containers:

    - name: nginx

      image: nginx:latest

    ports:

      - containerPort: 80

```

1. kind: Deployment

Hiermee geef je aan dat het object een Deployment is.

2. name: nginx-deployment

Dit is de unieke naam voor de Deployment binnen de namespace. Gebruik een geldige DNS-naam zonder spaties of speciale tekens.

3. replicas: 2

Hiermee geef je aan hoeveel exemplaren van de pod je wilt draaien. Hier zijn dat er 2, wat zorgt voor redundantie en eventueel high availability.

4. template:

Dit is een pod-template. Het beschrijft hoe elke pod eruit moet zien die door de Deployment wordt gemaakt. Hier definieer je onder andere:

- Labels die bepalen welke pods bij deze deployment horen.
- De container(s) met naam, image en poortinstellingen.

In Kubernetes hoef je niet per se een apart manifest voor een Pod te maken als je al een deployment manifest hebt. Een deployment bevat namelijk een pod-template dat beschrijft hoe elke pod eruit

moet zien. Kubernetes gebruikt deze template om automatisch de juiste Pods te creëren en te beheren.

In onderstaand stappenplan gaan we aan de slag met de basis deployment file zoals hierboven uitgelegd:

1. Verwijder alle clusters.

```
student@virt-k8s-XX:~$ k3d cluster list  
student@virt-k8s-XX:~$ k3d cluster delete ...
```

2. Maak nieuwe cluster aan voor de test.

```
student@virt-k8s-XX:~$ k3d cluster create mycluster --agents 1
```

3. Vervolgens maken we een yaml file aan met de inhoud van bovenstaand voorbeeld.

```
student@virt-k8s-XX:~$ nano dep.yaml  
  
apiVersion: apps/v1  
  
kind: Deployment  
  
metadata:  
  
  name: nginx-deployment  
  
spec:  
  
  replicas: 2  
  
  selector:  
  
    matchLabels:  
  
      app: nginx  
  
  template:  
  
    metadata:  
  
      labels:  
  
        app: nginx  
  
    spec:  
  
      containers:  
  
        - name: nginx  
  
          image: nginx:latest  
  
          ports:  
  
            - containerPort: 80
```

4. Als laatste stap volstaat het om deze nieuw aangemaakt manifest file te deployen.

```
student@virt-k8s-XX:~$ kubectl apply -f dep.yaml
deployment.apps/nginx-deployment created
```

Ten alle tijden kan men de status van een bepaalde deployment nagaan. Dit doen we door gebruik te maken van volgende cmd:

```
student@virt-k8s-XX:~$ kubectl get deployment nginx-deployment
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   2/2      2           2          90s
```

Zoals je kan zien hebben we hier een duidelijk overzicht van de status van de deployment met een aantal belangrijke statistieken:

- READY = geeft aan hoeveel pods er reeds klaar zijn van het totaal aantal replica's dat we hebben gespecificeerd.
- UP-TO-DATE = Welke pods de laatste image hebben volgens de manifest file (later zien we meer over updates via een deployment)
- AVAILABLE = hoeveel pods er beschikbaar zijn.
- AGE = hoelang de deployment reeds aan het draaien is (vanaf eerste apply)

In onze manifest file hebben we aangegeven dat we 2 replica's willen voor deze deployment. Dit kunnen we controleren op volgende manier via de pods:

```
student@virt-k8s-XX:~$ kubectl get pod
NAME           READY   STATUS    RESTARTS   AGE
nginx-deployment-54b9c68f67-8sqwj   1/1     Running   0          4m3s
nginx-deployment-54b9c68f67-nfhlz   1/1     Running   0          4m3s
```

11.9.3.2 Self-healing

Een van de belangrijkste verantwoordelijkheden van een deployment was de self-healing functie van de pods. Dit wil zeggen dat wanneer een bepaalde pod niet meer correct draait, de deployment processen er automatisch voor zullen zorgen dat we zo snel als mogelijk terug in de state zitten die in de manifest file staat aangegeven. In dit geval 2 replica's. Deze functionaliteit kunnen we ook zelf uittesten door manueel een pod te verwijderen. K8s zal automatisch een nieuwe pod aanmaken.

```
student@virt-k8s-XX:~$ kubectl delete pod nginx-deployment...
pod "nginx-deployment-54b9c68f67-8sqwj" deleted
```

```
student@virt-k8s-XX:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-54b9c68f67-m6f9h	1/1	Running	0	21s
nginx-deployment-54b9c68f67-nfh1z	1/1	Running	0	8m10s

Zoals je duidelijk kunt zien, heeft de self-healing-functie ingegrepen en automatisch een nieuwe pod aangemaakt.

11.9.3.3 Aanpassen deployment

Het grote voordeel van een Deployment is dat je altijd de gewenste staat (desired state) van je applicatie kan aanpassen. Dit kan op twee manieren:

- Imperatief, door rechtstreeks commando's te gebruiken (bijvoorbeeld via kubectl scale).
- Declaratief, door de manifest file aan te passen en deze opnieuw toe te passen met kubectl apply.

Gebruik bij voorkeur altijd de declaratieve manier.

Als je namelijk imperatieve commando's gebruikt, pas je de deployment aan in de cluster, maar niet de manifest file. Hierdoor ontstaat er een mismatch: bij een volgende apply van de manifest file kan je deployment teruggezet worden naar een oude configuratie.

De gouden regel is daarom om altijd veranderingen in de manifest file door te voeren en deze opnieuw te apply'en.

Als je bijvoorbeeld het aantal replica's wil wijzigen van 2 naar 6 pas je dit aan in de manifest file en voer je daarna "kubectl apply" uit. Kubernetes herkent aan de hand van de naam dat het dezelfde deployment is en voert alleen de gewijzigde instellingen door.

```
student@virt-k8s-XX:~$ kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	2/2	2	2	16m

We zullen nu instellen dat er 6 replica's moeten zijn.

```
student@virt-k8s-XX:~$ nano dep.yaml
```

...

spec:

replicas: 6

...

We voeren de wijzigingen door.

```
student@virt-k8s-XX:~$ kubectl apply -f dep.yaml  
deployment.apps/nginx-deployment configured
```

We checken de nieuwe situatie.

```
student@virt-k8s-XX:~$ kubectl get deployment  
  
NAME           READY   UP-TO-DATE   AVAILABLE   AGE  
nginx-deployment   6/6      6            6           20m
```

```
student@virt-k8s-XX:~$ kubectl get pods  
  
NAME                               READY   STATUS    RESTARTS   AGE  
nginx-deployment-54b9c68f67-dpc98   1/1     Running   0          2m16s  
nginx-deployment-54b9c68f67-g2q9m   1/1     Running   0          2m16s  
nginx-deployment-54b9c68f67-j57vx   1/1     Running   0          2m16s  
nginx-deployment-54b9c68f67-l2srk   1/1     Running   0          2m16s  
nginx-deployment-54b9c68f67-m6f9h   1/1     Running   0          13m  
nginx-deployment-54b9c68f67-nfh1z   1/1     Running   0          21m
```

11.9.3.4 Verwijderen deployment

Indien je geen gebruik meer wenst te maken van een bepaalde deployment, kan men deze verwijderen door gebruik te maken volgend cmd:

```
student@virt-k8s-XX:~$ kubectl delete deployment nginx-deployment  
deployment.apps "nginx-deployment" deleted  
  
student@virt-k8s-XX:~$ kubectl get deployment  
  
No resources found in default namespace.
```

Het kan even duren voor de pods weg zijn.

```
student@virt-k8s-XX:~$ kubectl get pods  
  
No resources found in default namespace.
```

Men kan ook een deployment aan de hand van de manifest file verwijderen:

```

student@virt-k8s-XX:~$ kubectl apply -f dep.yaml
deployment.apps/nginx-deployment created

student@virt-k8s-XX:~$ kubectl get deployment
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   6/6      6           6          14s

```

```

student@virt-k8s-XX:~$ kubectl delete -f dep.yaml
deployment.apps "nginx-deployment" deleted

```

Het verwijderen van een deployment is definitief. Alle pods worden verwijderd. Let dus steeds goed op dat je de juiste deployment verwijderd.

11.9.4 Rolling updates

We zullen eerst een agent toevoegen aan de cluster mycluster.

```

student@serverXX:~$ k3d node create agent-02 --role agent --
cluster mycluster
INFO[0000] Adding 1 node(s) to the runtime local cluster
'mycluster'...
INFO[0000] Using the k3d-tools node to gather environment
information
INFO[0000] Starting new tools node...
INFO[0000] Starting node 'k3d-mycluster-tools'
INFO[0000] HostIP: using network gateway 172.18.0.1 address
INFO[0000] Starting node 'k3d-agent-02-0'
INFO[0003] Successfully created 1 node(s)!

student@serverXX:~$ k3d cluster list
NAME          SERVERS   AGENTS   LOADBALANCER
mycluster     1/1       2/2      true

```

Naast het aanpassen van het aantal replica's voor de self-healing functionaliteit, heeft men ook de mogelijkheid om aan de hand van een deployment rolling updates uit te voeren. Dit is een manier van updaten waarbij men ervoor zorgt dat de microservice nooit down zal gaan. Er wordt maar

steeds een klein aantal van de replica's geüpdatet. Zo blijft men doorgaan tot de hele deployment is geüpdatet. Om dit mogelijk te maken in een deployment moet men een manifest file (noem die dep-rolling.yaml) aanmaken met bepaalde instellingen:

```
apiVersion: apps/v1
kind: Deployment (1)
metadata: (2)
  name: nginx-deployment
  annotations:
    kubernetes.io/change-cause: "Initial release"
spec:
  replicas: 10 (3)
  selector: (4)
    matchLabels:
      app: nginx
  revisionHistoryLimit: 5 (5)
  progressDeadlineSeconds: 300 (6)
  minReadySeconds: 10 (7)
  strategy: (8)
    type: RollingUpdate
    rollingUpdate:
      maxUnavailable: 1 (9)
      maxSurge: 1 (10)
  template:
    metadata:
      labels:
        app: nginx
  spec:
```

```
  containers:
    - name: nginx
      image: nginx:1.24
      ports:
        - containerPort: 80
```

1. kind: Deployment

Dit geeft aan dat het een Deployment-resource betreft.

2. Metadata:

Hierin staan metadata over het Deployment-object, zoals de naam en eventuele labels of annotaties voor identificatie en management. Meer uitleg hier later over.

3. replicas: 10

Het gewenste aantal gelijke pods dat altijd actief moet zijn. Hier zijn dat er 10.

4. Selector:

Dit is een label-selector die bepaalt welke pods onder beheer van deze Deployment vallen. De labels in .spec.template.metadata.labels moeten hieraan matchen.

5. revisionHistoryLimit: 5

Het aantal oude ReplicaSets dat wordt bewaard om terug te kunnen rollen naar een vorige versie.

6. progressDeadlineSeconds: 300

De maximale tijd in seconde waarin een update als succesvol moet worden beschouwd. Lukt dat niet, dan slaat de Deployment status met een foutmelding.

7. minReadySeconds: 10

Hoe lang een pod "ready" moet zijn voordat deze als beschikbaar wordt gezien en bijvoorbeeld oude pods worden verwijderd.

8. strategy:

De update-strategie die voor de Deployment wordt gebruikt. RollingUpdate zorgt voor gefaseerde updates met minimale downtime.

9. maxUnavailable: 1

Het maximum aantal pods dat tijdens een update tijdelijk niet beschikbaar mag zijn (hier 1).

10. maxSurge: 1

Het maximum aantal extra pods dat tijdelijk boven het gewenste replica-aantal mag draaien tijdens een update (hier 1).

Alvorens we deze manifest toe te passen is het noodzakelijk om even wat dieper in te gaan op de rollingupdate strategie. Hierin hebben we gespecificeerd dat we maar maximum 1 pod willen hebben die boven de desired state van 10 replica's is aan de hand van de maxSurge parameter. Dit wil zeggen dat er dus nooit meer dan 11 pods tegelijk mogen draaien tijdens het update proces. Daarnaast hebben we de maxUnavailable parameter aangegeven dat we nooit minder dan 9 pods willen hebben tijdens de update. De combinatie van deze twee parameters zal ervoor zorgen dat er maximum 2 pods tegelijk zullen worden geüpdatet.

Indien je je manifest aanpast met bovenstaande wijzigingen zal de deployment gewoon naar 10 replica's worden geschaald. Er zullen echter geen updates worden uitgevoerd daar de image versie nog steeds dezelfde is in container template. De updates zullen enkel uitgevoerd worden volgens de aangegeven strategie indien de image versie aangepast wordt of er een andere image voor deze deployment wordt gebruikt.

Wij gaan er van uit dat de deployment correct verloopt en er 10 replica's in onze deployment draaien met versie 1.0:

```
student@virt-k8s-XX:~$ kubectl apply -f dep-rolling.yaml
student@virt-k8s-XX:~$ kubectl get deployment
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   0/10     10          0           15s
```

Na enige tijd...

```
student@virt-k8s-XX:~$ kubectl get deployment
NAME           READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   10/10    10          10          30s
```

We checken nu of de pods gestart zijn.

```
student@virt-k8s-XX:~$ kubectl get pods
NAME                           READY   STATUS    RESTARTS   AGE
nginx-deployment-64dd99d679-2z8zd   1/1     Running   0          45s
...
nginx-deployment-64dd99d679-zqpfq   1/1     Running   0          45s
```

Om het update process nu correct te starten gaan we de manifestfile aanpassen naar een nieuwere image. Pas hiervoor dep-rolling.yaml aan.

```
image: nginx:1.26
```

Daarnaast gaan we ook bij de metadata in de manifest file de annotation aanpassen hetgeen we later kunnen gebruiken om eventueel snel terug te draaien naar een vorige versie. Pas de manifest file als volgt aan:

```
kubernetes.io/change-cause: "Update naar nginx:1.26"
```

Omdat we nu een aanpassing hebben gemaakt aan de manifest file gaan we deze terug apply'en.

```
student@virt-k8s-XX:~$ kubectl apply -f dep-rolling.yaml  
deployment.apps/nginx-deployment configured
```

De updates die nu gestart zijn, kunnen we volgen. Hiervoor kan je gebruik maken van het cmd "kubectl rollout status deployment". Dit zal op de console steeds aangeven wat de status is van de rollout. Om dit te stoppen volstaat het om "CTRL+C" in te drukken.

```
student@virt-k8s-XX:~$ kubectl rollout status deployment nginx-deployment
```

```
Waiting for deployment "nginx-deployment" rollout to finish: 2  
out of 10 new replicas have been updated...
```

```
Waiting for deployment "nginx-deployment" rollout to finish: 2  
out of 10 new replicas have been updated...
```

```
Waiting for deployment "nginx-deployment" rollout to finish: 2  
out of 10 new replicas have been updated...
```

```
Waiting for deployment "nginx-deployment" rollout to finish: 1  
old replicas are pending termination...
```

```
...
```

```
deployment "nginx-deployment" successfully rolled out
```

We tonen toont een overzicht van alle deployments in de huidige namespace van je Kubernetes cluster.

```
student@virt-k8s-XX:~$ kubectl get deployment
```

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	10/10	10	10	9m16s

In dit voorbeeld is het gehele update proces snel gedaan. Echter sommige updates kunnen heel wat tijd in beslag nemen, zeker wanneer je met grote deployments bezig bent. Daarom kan het soms noodzakelijk zijn om een rolling update even te pauzeren en terug te starten. Dit doe je door gebruik te maken van volgende twee cmd's:

```
kubectl rollout pause deploy <deployment-name>
```

```
kubectl rollout resume deploy <deployment-name>
```

11.9.5 Rollback

Het kan altijd voorkomen dat een bepaalde update voor problemen zorgt. In dat geval is het vaak noodzakelijk dat we een rollback doen naar een vorige versie. Dit kunnen we zoals altijd op 2 manieren doen:

- Imperatieve wijze (door gebruik van cmd's)
- Declaratieve wijze (door de manifest file terug aan te passen)

11.9.5.1 Imperatieve manier

Indien er echt een heel snelle rollback moet gebeuren, kan je altijd terug draaien naar een vorige versie door gebruik te maken van het kubectl rollout undo deployment cmd.

Alvorens we dit commando kunnen gebruiken, moeten we eerst achterhalen naar welke revision (replicaset) we de deployment willen terugdraaien. Hiervoor kan je gebruik maken van het kubectl rollout history deployment cmd.

```
student@virt-k8s-XX:~$ kubectl rollout history deployment nginx-deployment  
deployment.apps/nginx-deployment  
  
REVISION  CHANGE-CAUSE  
  
1          Initial release  
  
2          Update naar nginx:1.26
```

Nu wordt ook meteen duidelijk waarom we in de manifest file gebruik hebben gemaakt van de annotations. Deze geven ons bij het history cmd een duidelijk beeld van wat er in een bepaalde revision is gebeurd. Met dit commando zien we nu duidelijk dat we over 2 revisions beschikken en dus moeten terugdraaien naar revision 1. Om dit te bekomen volstaat het om het kubectl rollout undo deployment verder aan te vullen met de correcte revision voor de desbetreffende revision.

```
student@virt-k8s-XX:~$ kubectl rollout undo deployment nginx-deployment --to-revision=1  
deployment.apps/nginx-deployment rolled back
```

De opdracht kubectl rollout history deployment nginx-deployment toont een overzicht van alle revisies (versies) van het opgegeven deployment, inclusief de wijzigingsredenen.

```
student@virt-k8s-XX:~$ kubectl rollout history deployment nginx-deployment
deployment.apps/nginx-deployment

REVISION  CHANGE-CAUSE
2          Update naar nginx:1.26
3          Initial release
```

We tonen nu een lijst van alle pods in de huidige namespace, inclusief hun status, aantal containers dat draait, herstarts en leeftijd (tijdens rollback).

```
student@virt-k8s-XX:~$ kubectl get pods
```

NAME	READY	STATUS	RESTARTS	AGE
nginx-deployment-64dd99d679-585f5	1/1	Running	0	11s
nginx-deployment-64dd99d679-9mkgc	1/1	Running	0	23s
nginx-deployment-64dd99d679-b5s4h	1/1	Running	0	11s
nginx-deployment-64dd99d679-cx629	1/1	Running	0	34s
nginx-deployment-64dd99d679-gjrmgb	1/1	Running	0	23s
nginx-deployment-64dd99d679-jnjpz	1/1	Running	0	45s
nginx-deployment-64dd99d679-jwlt7	0/1	ContainerCreating	0	0s
nginx-deployment-64dd99d679-k59tm	0/1	ContainerCreating	0	0s
nginx-deployment-64dd99d679-lbpvd	1/1	Running	0	45s
nginx-deployment-64dd99d679-wxc98	1/1	Running	0	34s
nginx-deployment-75578ddfcc-2fb1f	1/1	Running	0	10m
nginx-deployment-75578ddfcc-b6zxk	0/1	Completed	0	10m
nginx-deployment-75578ddfcc-r2z6l	1/1	Terminating	0	11m

Na enige tijd runnen alle pods uiteraard.

We tonen nu de replicaset.

```
student@virt-k8s-XX:~$ kubectl get rs
```

NAME	DESIRED	CURRENT	READY	AGE
Pagina 53				

nginx-deployment-64dd99d679	10	10	10	18m
nginx-deployment-75578ddfcc	0	0	0	11m

Door de twee laatste commando's zien we duidelijk de werking van de replicasets. Per update wordt er een replicaset gemaakt om zo een geschiedenis op te bouwen van revisions. In de manifest file hebben wij aangegeven dat we maar 5 revisions willen bijhouden. Hiermee bedoelen we unieke revisions die effectief een verandering teweeg hebben gebracht. Bij de eerste release is de eerste Replicaset opgebouwd, bij de tweede release is er een tweede replicaset opgebouwd met de nieuwe versies. Wanneer we echter een rollback hebben gedaan is nu geen derde nieuwe replicaset opgebouwd maar heeft K8s intern gewoon een rollback gedaan naar de vorige replicaset (of de replicaset waarnaar we hebben verwezen). Dit is zelfs correct gereflecteerd in de history. Daar zien we dat bij revision 3 we terug op de initial release zitten. Hiermee hebben een duidelijke kracht van het gebruik van de replicaset aangetoond.

Deze manier van werken geniet, zoals reeds aangehaald, niet de voorkeur. Door het gebruik van imperatieve cmd's wordt de manifest file niet aangepast en heeft dit nu een potentieel gevaar gevormd. Als een andere engineer nu de manifest file terug deployed is deze deployment helemaal niet in de desired state dat we wensen.

11.9.5.2 Declaratieve manier

Beter is om de versie aan te passen in de manifest file en terug te deployen. Dit zorgt ervoor dat het manifest steeds up to date is.

Pas de deployment file terug aan naar de oorspronkelijke situatie:

```
student@virt-k8s-XX:~$ nano dep-rolling.yaml
...
kubernetes.io/change-cause: "Initial release"
...
image: nginx:1.24
...
```

```
student@virt-k8s-XX:~$ kubectl apply -f dep-rolling.yaml
deployment.apps/nginx-deployment configured
```

```

student@virt-k8s-XX:~$ kubectl get deployment

NAME          READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   10/10     10           10          35m

student@virt-k8s-XX:~$ kubectl rollout history deployment nginx-
deployment.apps/nginx-deployment

REVISION  CHANGE-CAUSE
2          Update naar nginx:1.26
3          Initial release

```

De initiele deployment is weer aan het draaien. We passen de manifest file nu terug aan zodat we versie 2 aan het draaien zijn en apply'en de manifest file.

```

student@virt-k8s-XX:~$ nano dep-rolling.yaml

...
Kubernetes.io/change-cause: "Update naar nginx:1.26"
...
image: nginx:1.26
...

```

```

student@virt-k8s-XX:~$ kubectl apply -f dep-rolling.yaml
deployment.apps/nginx-deployment configured

```

```

student@virt-k8s-XX:~$ kubectl rollout history deployment nginx-
deployment
deployment.apps/nginx-deployment

REVISION  CHANGE-CAUSE

```

- 3 Initial release
- 4 Update naar nginx:1.26

Wacht nu even tot de update volledig is afgerond. Dit kan je altijd nagaan door de status van de deployment te raadplegen:

```
student@virt-k8s-XX:~$ kubectl get deployment  
NAME                READY   UP-TO-DATE   AVAILABLE   AGE  
nginx-deployment  10/10   10            10          13m
```

Pas nu de manifest file terug aan door zowel de image op nginx:1.24 te zetten en de annotation aan te passen met een passende boodschap vb: 'Rollback to nginx 1.24 due to bug'. Vervolgens de manifest file terug apply'en.

```
student@virt-k8s-XX:~$ nano dep-rolling.yaml  
...  
  Kubernetes.io/change-cause: "Rollback to nginx 1.24 due to  
  bug"  
...  
  image: nginx:1.24  
...  
student@virt-k8s-XX:~$ kubectl apply -f dep-rolling.yaml  
deployment.apps/nginx-deployment configured  
  
student@virt-k8s-XX:~$ kubectl rollout history deployment nginx-deployment  
deployment.apps/nginx-deployment  
REVISION  CHANGE-CAUSE  
4          Update naar nginx:1.26  
5          Rollback to nginx 1.24 due to bug
```

```
student@virt-k8s-XX:~$ kubectl get deployment

NAME           READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   10/10    10          10          17m
```

11.10 Storage

Containers hebben vaak een kort leven. Als een container crasht, start Kubernetes automatisch een nieuwe container om de app beschikbaar te houden.

De data van de vorige container gaat daarbij verloren, want elke container heeft zijn eigen geïsoleerd bestandssysteem.

Voorbeeld opslag in filesystem van container:

```
apiVersion: v1
kind: Pod
metadata:
  name: temp-data-demo
spec:
  containers:
  - name: app
    image: busybox
    command: ['sh', '-c', 'echo "data" > /tmp/file.txt && sleep 3600']
```

Wanneer we deze zouden deployen zal er data worden weggeschreven in de /tmp/file.txt. Echter zodra de pod sterft zal deze data niet meer bestaan en zal de nieuwe pod starten met een nieuwe geïsoleerd filesystem. Gegevens opgeslagen in eigen, privé filesystem van containers gaan altijd verloren als de container crasht of de pod opnieuw opstartt.

Dit zouden we kunnen tegengaan door gebruik te maken van tijdelijke volumes (ephemeral volumes). Nemen we bijvoorbeeld volgende pod:

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-volume
spec:
  containers:
  - name: app
    image: busybox
    command: ['sh', '-c', 'echo "data" > /data/file.txt && sleep 3600']
    volumeMounts:
```

```

      - name: my-volume
        mountPath: /data
    volumes:
      - name: my-volume
        emptyDir: {} # Simple empty directory that survives
        container restarts

```

In deze volume mount wordt het volume my-volume gekoppeld aan het pad /data binnen de container.

In de volumes sectie wordt my-volume aangemaakt als type emptyDir: {}.

Het emptyDir volume wordt aangemaakt zodra de Pod op een node wordt gestart. Aanvankelijk is het volume leeg.

Het volume blijft bestaan zolang de Pod actief is, ongeacht het starten of crashen van individuele containers binnen die Pod. Hierdoor blijft de data behouden bij containerrestarts, maar niet als de volledige Pod verwijderd wordt.

11.10.1 Soorten volumes

Kubernetes maakt onderscheid tussen container filesystem, tijdelijke volumes en persistente opslag. Het container filesystem is privé voor de container en verdwijnt bij een herstart. Tijdelijke volumes bestaan zolang de Pod leeft en verdwijnen zodra de Pod stopt, terwijl persistente opslag data bewaart onafhankelijk van de levenscyclus van de Pod.

- Container filesystem

Het standaard geïsoleerde bestandssysteem van een container. Data hierin gaat verloren zodra de container herstart, crasht of de Pod eindigt. Niet gedeeld tussen containers en niet geschikt voor persistente opslag.

- Tijdelijke volumes (ephemeral volumes)

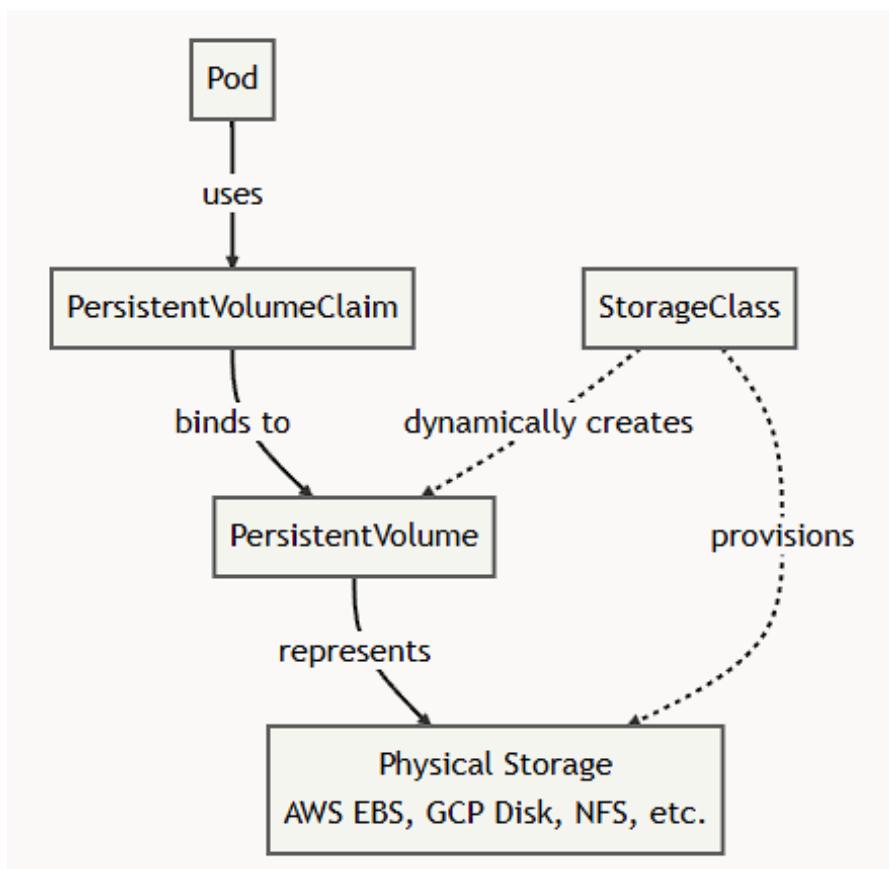
Tijdelijke volumes bestaan zolang de Pod leeft.

Ze worden beheerd door Kubernetes en zijn gedeeld tussen containers binnen dezelfde Pod. Ze verdwijnen volledig wanneer de Pod wordt verwijderd.

- emptyDir
 - Een lege, tijdelijke map die Kubernetes aanmaakt wanneer de Pod start.
 - De map wordt gedeeld tussen containers in de Pod en verdwijnt zodra de Pod stopt.
- configMap
 - Een volume dat configuratiegegevens bevat.
 - Beschikbaar als alleen-lezen bestanden voor containers.
 - Niet bedoeld voor langdurige opslag; data komt uit de ConfigMap-resource.
- Secret
 - Een volume dat gevoelige gegevens (zoals wachtwoorden of tokens) bevat.

Wordt als beveiligde, alleen-lezen bestanden aangeboden aan containers.

- **downwardApi**
Een volume waarmee Pod-informatie (bijvoorbeeld naam, labels, resource limits) als bestanden beschikbaar wordt gemaakt in de container.
Handig voor applicaties die metadata van de Pod nodig hebben.
- **Persistent storage**
 - **PersistentVolume (PV)**
Dit is een stuk opslagruimte dat door een beheerder is gecreëerd en beschikbaar is in het cluster, onafhankelijk van de levensduur van een Pod. Het vertegenwoordigt de fysieke opslagresource.
 - **PersistentVolumeClaim (PVC)**
Dit is het verzoek van een gebruiker of applicatie om toegang te krijgen tot een PersistentVolume met bepaalde eigenschappen zoals grootte en toegangsmodus.
 - **StorageClass**
Dit is een object waarmee de beheerder opslagklassen definieert met specifieke eigenschappen (zoals prestaties of back-upbeleid). Een StorageClass maakt dynamische provisioning van PersistentVolumes mogelijk, waardoor volumes automatisch worden aangemaakt wanneer een PVC wordt ingediend.
Een StorageClass is niet verplicht maar wel aanbevolen.



Deze drie objecten werken samen om ervoor te zorgen dat applicaties duurzaam en flexibel opslag kunnen gebruiken binnen een Kubernetes-cluster.

Opgelet: persistent data wordt verwijderd wanneer de cluster wordt verwijderd tenzij je bij het aanmaken van de cluster

11.10.2 PersistentVolume (PV)

Een persistent volume vertegenwoordigt de effectieve opslag in een cluster. Dit volume is onafhankelijk van een specifieke Pod, wat betekent dat de data in het Persistent Volume behouden blijft zelfs als de Pod die er gebruik van maakt stopt, verwijderd wordt of opnieuw wordt gestart.

Hieronder zien we een voorbeeld van een persistent volume

```
apiVersion: v1
kind: PersistentVolume (1)
metadata:
  name: my-pv (2)
spec:
  capacity:
    storage: 10Gi (3)
  accessModes:
    - ReadWriteOnce (4)
  persistentVolumeReclaimPolicy: Retain (5)
  hostPath:
    path: /mnt/data (6)
```

1. kind: PersistentVolume

We definiëren een Persistent volume.

2. name: my-pv

De naam van het volume is my-pv.

3. storage: 10Gi

Het volume stelt 10 gigabyte opslagruimte beschikbaar.

4. ReadWriteOnce

Dit betekent dat het volume door één node tegelijk voor lezen en schrijven kan worden gebruikt.

5. persistentVolumeReclaimPolicy: Retain

Dit bepaalt dat de opslag behouden blijft nadat de PersistentVolumeClaim (PVC) losgekoppeld is; data wordt dus niet automatisch verwijderd.

6. path: /mnt/data

Dit volume maakt gebruik van een fysieke map op de node zelf als opslag. Dit is alleen geschikt voor testomgevingen of eenvoudige setups, want bij productiesystemen worden meestal netwerk- of cloudopslag gebruikt.

11.10.3 PersistentVolumeClaim (PVC)

Een PersistentVolumeClaim is eigenlijk een verzoek om toegang te krijgen tot een bepaald volume. Hierbij zal men ook steeds een access mode moeten aangeven alsook laten weten hoeveel ruimte we effectief willen gebruiken van dat bepaald volume.

```
apiVersion: v1
kind: PersistentVolumeClaim (1)
metadata:
  name: my-pvc (2)
spec:
  accessModes:
    - ReadWriteOnce (3)
  resources:
    requests:
      storage: 8Gi (4)
```

1. kind: PersistentVolumeClaim
We definiëren een PVC.
2. name: my-pvc
De naam van deze aanvraag voor opslag.
3. ReadWriteOnce
Dit geeft aan dat de opslag later slechts door één node tegelijk gelezen en beschreven mag worden.
4. storage: 8Gi
De PVC opslagruijnte vraagt minimaal 8 GiB.

Met bovenstaande volumeClaim kan je nu aan de slag om te gebruiken in bijvoorbeeld een pod:

```
apiVersion: v1
kind: Pod (1)
metadata:
  name: my-pod (2)
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - name: storage (3)
          mountPath: /usr/share/nginx/html (4)
  volumes:
    - name: storage
      persistentVolumeClaim:
        claimName: my-pvc (5)
```

1. kind: Pod
Dit beschrijft, zoals jullie weten, een Pod.
2. name: my-pod
De naam van de Pod, namelijk "my-pod".
3. name: storage
Geeft aan dat een volume met de naam "storage" binnen de container wordt gekoppeld.
4. mountPath: /usr/share/nginx/html
Dit is de locatie in de container waar het volume wordt gemount, in dit geval de webroot voor Nginx.
5. claimName: my-pvc
Een PersistentVolumeClaim met de naam "my-pvc" wordt gebruikt.

11.10.4 StorageClass

Een StorageClass maakt het mogelijk om persistent volumes automatisch te creëren wanneer je een PersistentVolumeClaim aanmaakt. Dit concept wordt vooral veel gebruikt in cloudomgevingen. Cloudomgevingen vallen buiten de doelstellingen van dit hoofdstuk.

Bij k3s is standaard al een default StorageClass aanwezig (zie verder), genaamd local-path, die de lokale opslag van de nodes gebruikt.

11.10.5 Access modes

Persistent volumes kunnen drie verschillende toegangsniveaus (access modes) hebben:

- ReadWriteOnce (RWO)
Volume kan gemount worden in Read-Write mode, maar enkel door 1 node.
- ReadOnlyMany (ROX)
Volume kan gemount worden als Read-Only door meerdere nodes.
- ReadWriteMany (RWX)
Volume kan gemount worden als Read-Write door meerdere nodes.

In de praktijk hangt de beschikbare access mode sterk af van het type opslag (storage) dat achter het volume zit. Niet elk opslagtype ondersteunt bijvoorbeeld gelijktijdige toegang door meerdere nodes.

11.10.6 Reclaim Policies

Bij het verwijderen van een PersistentVolume (PV) bepaalt de persistentVolumeReclaimPolicy wat er met de bijbehorende storage gebeurt. Er zijn twee mogelijke opties:

- Delete: de storage wordt automatisch verwijderd wanneer het PV wordt verwijderd (dit is de standaardoptie voor PV's die via een StorageClass automatisch worden aangemaakt)..
- Retain: de storage blijft bestaan, ook nadat de PV is verwijderd.

Voorbeeld in een PV-configuratie:

```
persistentVolumeReclaimPolicy: Retain
```

Opgelet: deze instelling geldt voor alle PV's: zowel degenen die handmatig als via storageclass zijn aangemaakt.

11.10.7 Voorbeelden

Om de verschillende types van storage duidelijk te maken, gaan we aan de slag met een nieuwe cluster. Om zoveel mogelijk resources te besparen, verwijder je best alle andere clusters. Zodra je dat gedaan hebt, kan je aan de slag om een nieuwe cluster aan te maken.

```
student@virt-k8s-XX:~$ k3d cluster list
student@virt-k8s-XX:~$ k3d cluster delete ...
student@virt-k8s-XX:~$ k3d cluster create storage-lab --agents 2
INFO[0000] Prep: Network
INFO[0000] Created network 'k3d-storage-lab'
```

```
student@virt-k8s-XX:~$ kubectl get nodes
```

NAME	STATUS	ROLES	AGE	VERSION
k3d-storage-lab-agent-0	Ready	<none>	80s	v1.31.5+k3s1
k3d-storage-lab-agent-1	Ready	<none>	80s	v1.31.5+k3s1
k3d-storage-lab-server-0	Ready	control-plane, master	83s	v1.31.5+k3s1

We bekijken eerst de default storageclass is bij de cluster:

```
student@virt-k8s-XX:~$ kubectl get storageclass
NAME          PROVISIONER           RECLAIMPOLICY  ...
local-path (default)  rancher.io/local-path  Delete        ...
```

Het gaat hier om een lokale opslag StorageClass die dynamische provisioning voorziet binnen het cluster. Dit is typisch in test- of developmentomgevingen met lokale opslag.

```
student@virt-k8s-XX:~$ kubectl get storageclass local-path -o yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  annotations:
    defaultVolumeType: local
```

...

Dit wil zeggen dat wanneer er een PVC gemaakt wordt zonder StorageClassName, de default storageclass zal gebruikt worden.

11.10.7.1 Opslag in container filesystem

Soms kan het handig zijn om specifiek vluchtige storage te gebruiken. Hiervoor moeten we eerst even bestuderen hoe dit effectief in zijn werk gaat.

Maak een nieuwe manifest file (temptest.yaml) aan:

```
apiVersion: v1
kind: Pod
metadata:
  name: temp-data-pod
spec:
  containers:
    - name: writer
      image: busybox
      command: ['sh', '-c', 'while true; do echo "Log entry at $(date)" >> /tmp/app.log; sleep 5; done']
```

Deploy deze pod.

```
student@virt-k8s-XX:~$ kubectl apply -f temptest.yaml
kubectl apply -f temptest.yaml
```

Kijk naar de log file die wordt opgebouwd.

```
student@virt-k8s-XX:~$ kubectl exec temp-data-pod -- tail -f /tmp/app.log
Log entry at Sun Nov 16 11:32:56 UTC 2025
Log entry at Sun Nov 16 11:33:01 UTC 2025
^C
```

Bekijk de inhoud van de map /tmp/ binnen de container van de Pod temp-data--pod.

```
student@virt-k8s-XX:~$ kubectl exec temp-data-pod -- ls -la /tmp/
total 4
drwxrwxrwt    1 root     root            21 Nov  1 11:38 .
drwxr-xr-x    1 root     root            62 Nov  1 11:38 ..
```

```
-rw-r--r--    1 root      root        2310 Nov  1 11:43 app.log
```

Verwijder nu de pod.

```
student@virt-k8s-XX:~$ kubectl delete -f temptest.yaml  
pod "temp-data-pod" deleted
```

Start pod metzelfde instellingen opnieuw op.

```
student@virt-k8s-XX:~$ kubectl apply -f temptest.yaml  
pod/temp-data-pod created
```

Kijk terug naar de log file die wordt opgebouwd.

```
student@virt-k8s-XX:~$ kubectl exec temp-data-pod -- tail -f /tmp/app.log  
Log entry at Sun Nov 16 11:41:07 UTC 2025  
Log entry at Sun Nov 16 11:41:12 UTC 2025  
^C
```

Zoals je kan zien, wordt het bestand gevuld met nieuwe entry's (kijk naar het uur).

11.10.7.2 Persistent storage - Manueel

Het eerste dat we moeten doen, is een Persistent Volume aanmaken. Daarvoor maak je een YAML-bestand met de naam manual-pv.yaml, waarin je de specificaties van het volume definieert

```
apiVersion: v1  
kind: PersistentVolume  
metadata:  
  name: manual-pv  
spec:  
  capacity:  
    storage: 1Gi  
  accessModes:  
    - ReadWriteOnce  
  persistentVolumeReclaimPolicy: Retain  
  storageClassName: manual  
  hostPath:  
    path: /tmp/k3d-manual-pv
```

Daarna moet je ook een PersistentVolumeClaim manifest aanmaken. Wij geven het de naam manual-pvc.yaml

```
apiVersion: v1  
kind: PersistentVolumeClaim  
metadata:  
  name: manual-pvc
```

```

spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
  storageClassName: manual

```

Deze beide gaan we nu apply'en wat verder in detail bekijken.

```

student@virt-k8s-XX:~$ kubectl apply -f manual-pv.yaml
persistentvolume/manual-pv created
student@virt-k8s-XX:~$ kubectl apply -f manual-pvc.yaml
persistentvolumeclaim/manual-pvc created

```

We tonen nu een overzicht van alle Persistent Volumes (PV's) die in je Kubernetes-cluster zijn aangemaakt.

```

student@virt-k8s-XX:~$ kubectl get pv
  NAME      CAPACITY   ACCESS MODES   RECLAIM POLICY   ...
  manual-pv  1Gi        RWO           Retain          ...

```

We tonen nu een overzicht van alle PersistentVolumeClaims (PVC's) in de Kubernetes-cluster.

```

student@virt-k8s-XX:~$ kubectl get pvc
  NAME      STATUS    VOLUME   CAPACITY   ...
  manual-pvc Bound    manual-pv  1Gi        ...

```

We vragen nu meer details op over PVC manual-pvc.

```

student@virt-k8s-XX:~$ kubectl describe pvc manual-pvc
Name:           manual-pvc
Namespace:      default
StorageClass:   manual
Status:         Bound
Volume:         manual-pv
Labels:         <none>

```

```

Annotations:    pv.kubernetes.io/bind-completed: yes
                pv.kubernetes.io/bound-by-controller: yes
Finalizers:    [kubernetes.io/pvc-protection]
Capacity:      1Gi
Access Modes:  RWO
VolumeMode:   Filesystem
Used By:       <none>
Events:        <none>

```

Vervolgens gaan we deze PVC gebruiken in een pod. Hiervoor volstaat het om een pod manifest file aan te maken genaamd pod-with-pvc.yaml.

```

apiVersion: v1
kind: Pod
metadata:
  name: pvc-demo
spec:
  containers:
  - name: app
    image: busybox
    command: ['sh', '-c', 'echo "Persistent data" > /mydata/file.txt && sleep 3600']
  volumeMounts:
  - name: persistent-storage
    mountPath: /mydata
  volumes:
  - name: persistent-storage
    persistentVolumeClaim:
      claimName: manual-pvc

```

Dan rest je alleen nog de pod te apply'en en een file aan te maken.

```

student@virt-k8s-XX:~$ kubectl apply -f pod-with-pvc.yaml
pod/pvc-demo created

```

We schrijven de tekst "Important data" naar het bestand /mydata/important.txt binnenvin de Pod pvc-demo, zodat het wordt opgeslagen op de gekoppelde PersistentVolume.

```

student@virt-k8s-XX:~$ kubectl exec pvc-demo -- sh -c 'echo "Important data" > /mydata/important.txt'

```

We tonen nu de inhoud van het bestand /mydata/important.txt binnennin de container van de pod pvc-demo.

```
student@virt-k8s-XX:~$ kubectl exec pvc-demo -- cat  
/mydata/important.txt
```

Important data

We tonen nu de inhoud van het bestand /mydata/file.txt binnennin de Pod pvc-demo.

```
student@virt-k8s-XX:~$ kubectl exec pvc-demo -- cat  
/mydata/file.txt
```

Persistent data

Verwijder nu de pod.

```
student@virt-k8s-XX:~$ kubectl delete pod pvc-demo  
pod "pvc-demo" deleted
```

We maken nu terug eenzelfde pod aan.

```
student@virt-k8s-XX:~$ kubectl apply -f pod-with-pvc.yaml  
pod/pvc-demo created
```

We checken nu of de data nog beschikbaar is.

```
student@virt-k8s-XX:~$ kubectl exec pvc-demo -- cat  
/mydata/important.txt
```

Important data

```
student@virt-k8s-XX:~$ kubectl exec pvc-demo -- cat  
/mydata/file.txt
```

Persistent data

Zoals we constateren is dit inderdaad het geval!

We verwijderen nu alle resources van deze demo.

```
student@virt-k8s-XX:~$ kubectl delete pod pvc-demo  
student@virt-k8s-XX:~$ kubectl delete pvc manual-pvc  
student@virt-k8s-XX:~$ kubectl delete pv manual-pv
```

11.10.7.3 Persistent storage – Dynamische storage

In k3d kunnen we ook dynamische opslagprovisioning gebruiken. Dit gebeurt via de ingebouwde default StorageClass, meestal local-path, omdat er geen cloudprovider zoals AWS of GCP beschikbaar is om opslag te beheren.

We maken hiervoor eerst een nieuwe PVC-manifest genaamd dynamic-pvc.yaml:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: dynamic-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

StorageClassName is optioneel; als je het niet opgeeft, wordt in K3d standaard de local-path StorageClass gebruikt.

We maken ook een Pod manifest aan dynamic-pod.yaml die de PVC gebruikt:

```
apiVersion: v1
kind: Pod
metadata:
  name: dynamic-demo
spec:
  containers:
    - name: app
      image: nginx
      volumeMounts:
        - name: web-content
          mountPath: /usr/share/nginx/html
  volumes:
    - name: web-content
      persistentVolumeClaim:
        claimName: dynamic-pvc
```

We hebben geen persistent volume aangemaakt. Zoals aangehaald zal de local-path StorageClass gebruikt worden.

```
student@virt-k8s-XX:~$ kubectl get pv
```

```
No resources found
```

We deployen nu de persistent volume claim.

```
student@virt-k8s-XX:~$ kubectl apply -f dynamic-pvc.yaml
persistentvolumeclaim/dynamic-pvc created
```

We vragen nu persistent volume claim op.

```
student@virt-k8s-XX:~$ kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	...
dynamic-pvc	Pending				local-path	...

Je ziet bij STATUS Pending staan omdat Kubernetes wacht met binden tot er een Pod is die gebruik maakt van de persistent volume claim.

Uiteraard zijn er nu nog geen persistent volumes.

```
student@virt-k8s-XX:~$ kubectl get pv
```

No resources found

Dit zal immers pas aangemaakt worden wanneer er een pod gebruikt maakt van het PVC dat op zijn beurt automatisch een PersistentVolume zal aanmaken.

```
student@virt-k8s-XX:~$ kubectl apply -f dynamic-pod.yaml
```

pod/dynamic-demo created

We vragen nu terug persistent volumes claims en persistant volume op.

```
student@virt-k8s-XX:~$ kubectl get pvc
```

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	...
dynamic-pvc	Bound	pvc-...	2Gi	RWO	local-path	...


```
student@virt-k8s-XX:~$ kubectl get pv
```

NAME	CAPACITY	ACCESS MODES	RECLAIM POLICY	STATUS	...
pvc-...	2Gi	RWO	Delete	Bound	...

Doordat de pod gebruikt maakt van de PVC met de default storageclass, is er nu ook automatisch een Volume aangemaakt. Deze heeft een dynamische naam die aangemaakt is door het K8s zelf.

Nu kunnen we in de pod een bestand (webpagina) wegschrijven, dat door nginx zal gebruikt worden.

```
student@virt-k8s-XX:~$ kubectl exec -it dynamic-demo -- /bin/sh
```

```
# echo "<h1>Persistent Web Content</h1>" >
/usr/share/nginx/html/index.html
```

```
# exit
```

We starten nu port forward.

```
student@virt-k8s-XX:~$ kubectl port-forward pod/dynamic-demo  
8080:80 &
```

...

We vragen nu webpagina op vanaf de host.

```
student@virt-k8s-XX:~$ curl localhost:8080  
Handling connection for 8080  
<h1>Persistent Web Content</h1>
```

Wanneer we nu de pod verwijderen zal de storage nog steeds blijven bestaan.

Verwijder nu de pod.

```
student@virt-k8s-XX:~$ kubectl delete pod dynamic-demo  
pod "dynamic-demo" deleted
```

Het opvragen van de webpagina gaat nu uiteraard niet meer.

```
student@virt-k8s-XX:~$ curl localhost:8080  
Handling connection for 8080  
...  
curl: (52) Empty reply from server  
error: lost connection to pod
```

We maken nu terug een nieuwe pod aan metzelfde instellingen.

```
student@virt-k8s-XX:~$ kubectl apply -f dynamic-pod.yaml  
pod/dynamic-demo created
```

We starten nu port forward opnieuw op.

```
student@virt-k8s-XX:~$ kubectl port-forward pod/dynamic-demo  
8080:80 &
```

...

We vragen nu de webpagina opnieuw op.

```
student@virt-k8s-XX:~$ curl localhost:8080  
Handling connection for 8080
```

<h1>Persistent Web Content</h1>

We verwijderen de nieuwe pod.

```
student@virt-k8s-XX:~$ kubectl delete pod dynamic-demo  
pod "dynamic-demo" deleted
```

Wanneer we de Claim verwijderen zal het volume nu ook automatisch verwijderd worden. Dit komt omdat de reclaimPolicy op delete staat voor local-path. Hieronder zie je dat duidelijk staan.

```
student@virt-k8s-XX:~$ kubectl get storageclass  
  
NAME           PROVISIONER          RECLAIMPOLICY ...  
local-path (default)  rancher.io/local-path  Delete ...
```

We verwijderen persistent volume claim dynamic-pvc.

```
student@virt-k8s-XX:~$ kubectl delete pvc dynamic-pvc  
persistentvolumeclaim "dynamic-pvc" deleted
```

We vragen nu de persistent volumes op.

```
student@virt-k8s-XX:~$ kubectl get pv  
  
No resources found
```

Zoals verwacht is er geen persistent volume meer.

11.11 Disclaimer

Het hoofdstuk van K8s is gebaseerd op het werk van Tom Cool (lector bachelor) aangevuld met eigen inbreng (ook gebruik makend van het internet).