xamira networks

Kubernetes

Hands-On Training Container Orchestration

Version 1.2.0, 2023-06-12

xamira networks

1. Command Overview

This overview contains useful example commands, it is not a comprehensive reference!

1.1. Useful `kubectl` Commands

1.1.1. Enabling Bash Completion

```
echo "source <(kubectl completion bash)" >> ~/.bashrc
```

1.1.2. Cluster and API Information

```
kubectl cluster-info

kubectl api-versions

kubectl api-resources
```

1.1.3. Creating, Displaying and Deleting Objects

```
kubectl explain Pod.spec

kubectl apply -f file.yaml

kubectl get pods

kubectl get pods -w

kubectl get pods -o wide

kubectl get deploy,rs,pods

kubectl get pod/podname -o yaml

kubectl get all

kubectl describe pod/podname

kubectl delete pod/podname

kubectl delete pod/podname --grace-period=0
```

1.1.4. Handling Labels

```
kubectl get pods -l app=training --show-labels

kubectl get pods -l 'env in (prod,dev)' --show-labels

kubectl label pods --overwrite env=prod -l env=dev
```

1.1.5. Interacting with Pods and Containers

```
kubectl exec -ti podname -c containername -- /bin/bash

kubectl logs podname -c containername -f
```

1.1.6. Modifying Existing Objects

```
kubectl edit pod/podname

kubectl patch pod/podname -p '{"spec":{"containers":[{"name":"httpd","image":"nginx"}]}}'
```

1.1.7. Updating and Scaling

```
kubectl scale deployment/httpd --replicas=10

kubectl set image deployment/httpd httpd=nginx:latest --record=true

kubectl rollout status deployment/httpd

kubectl rollout pause deployment/httpd

kubectl rollout resume deployment/httpd

kubectl rollout history deployment/httpd

kubectl rollout undo deployment/httpd --to-revision=2 --record=true
```

1.1.8. Create Objects by Command

```
kubectl run webserver1 --image=nginx

kubectl run webserver2 --image=httpd --labels app=training --restart=Never

kubectl expose deployment/webserver1 --port=80

kubectl create secret generic dbcreds --from-literal=password=root
```

1.2. Useful `stern` Commands

```
source <(stern --completion=bash)

stern --all-namespaces -l app=training

stern training -o json | jq .

stern --template '{{.Message}} ({{.Namespace}}/{{.PodName}}/{{.Container-
Name}})' training
```

1.3. Useful `helm` Commands

1.3.1. Managing Helm Repositories

```
helm repo add example http://charts.example.com/

helm repo list

helm search repo example
```

1.3.2. Examining Helm Charts

```
helm show readme example/example

helm show values example/example
```

1.3.3. Installing Helm Charts and Managing Helm Releases

```
helm install -n app1 application example/example

helm install -n app1 application example/example --set ingress.enable=true

helm install -n app1 application example/example -f app1-values.yaml

helm list -n app1

helm upgrade -n app1 application example/example --version=1.2.0

helm uninstall -n app1 application
```

2. Simple Pod

2.1. Explanations

A Pod is the smallest unit in which containers can be run on Kubernetes.

A Pod can contain one or multiple containers. The containers of a Pod have a special relationship:

- All containers in a Pod share the same network and IPC namespaces in the Linux kernel.
- All containers in a Pod run on the same Kubernetes Node, since they would otherwise not be able to share kernel namespaces.
- The volumes of a Pod can be mounted in all containers of the Pod, to allow data exchange between the containers.

The shared network namespace has further implications for containers running in the same pod:

- All containers see the same network interfaces with the same IP addresses and routes.
- Each TCP and UDP port can be opened by only one container at a time.
- The containers can communicate with each other via localhost.

The question, if two containers should be bundled into one Pod or better be launched in two separate Pods is an important decision. An answer can often be found in the question, whether or not containers should be scaled independently. In the case of an application server and its associated database, the question is usually "yes" — on the one hand, because the resource requirements and thus the number of required instances of the two containers can be very different, and on the other hand, because additional configurations is required for the vertical scaling of a database server. The situation is different, however, for a monitoring agent which runs next to an application server to make data related to the application server available for monitoring. In this case, it is actually desirable to collect data from each server individually, which is why these two containers should always be run and scaled in one Pod.

2.2. Example: Pod

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: training-webserver-shell
  labels:
    app: training
    component: webserver
spec:
  containers:
    - name: webserver
      image: httpd:latest
    - name: shell1
      image: alpine:latest
      command:
        - /bin/sh
        - -c
      args:
        - while true; do date; sleep 1; done
    - name: shell2
      image: debian:latest
      tty: true
      stdin: true
```

3. Non-Persistent Volumes

3.1. Explanations

Non-persistent volumes, short NPV, are sometimes also referred to as ephemeral volumes and allow data to be exchanged between the containers in a Pod. A volume of the type `emptyDir` is used for this purpose. This is simply a directory that the kubelet on the respective node creates when the container is started. emptyDir volumes can only be used within a Pod; they do, however, not allow data exchange with containers of other Pods. Furthermore, emptyDir volumes are deleted when the associated Pod is terminated.

3.2. Example: Pod with Non-Persistent Volume

```
apiVersion: v1
kind: Pod
metadata:
  name: training-webserver-npv
  labels:
    app: training
    component: webserver
spec:
  containers:
    - name: webserver
      image: httpd:latest
      volumeMounts:
        - mountPath: /usr/local/apache2/htdocs
          name: htdocs
    - name: shell
      image: alpine:latest
      volumeMounts:
        - mountPath: /mnt
          name: htdocs
  volumes:
    - name: htdocs
      emptyDir: {}
```

4. Pods with Init Containers

4.1. Explanations

Init containers are defined similarly to the regular containers of a Pod. However, when the Pod starts, the init containers are executed first. This happens sequentially in the order in which the init containers are defined. Only when an init container terminates successfully, the next init container is started and only when the last init container is terminated the regular containers are executed.

Init containers can be used, for example, to initialize applications, to customize configuration files, or to register a Pod with a service for service discovery. Data exchange is usually implemented using non-persistent volumes which are mounted in the init containers as well as in the regular containers.

4.2. Example: Pod with Init Container

```
apiVersion: v1
kind: Pod
metadata:
  name: training-webserver-init
  labels:
    app: training
    component: webserver
spec:
  initContainers:
    - name: greeter
      image: alpine:latest
      command:
        - /bin/sh
        - -c
      args:
        - echo 'Hello!' > /mnt/index.html
      volumeMounts:
        - mountPath: /mnt
          name: htdocs
  containers:
    - name: webserver
      image: httpd
      volumeMounts:
        - mountPath: /usr/local/apache2/htdocs
          name: htdocs
  volumes:
    - name: htdocs
      emptyDir: {}
```

5. Persistent Volumes

5.1. Explanations

Persistent volumes allow permanently storing data independently of Pods.

Objects of the kind PersistentVolume describe the specific location where data is stored. This could, for example, be a specific share on an NFS server, a LUN on an iSCSI target, a VMDK file in a vSphere datastore, or a volume in a cloud provider's block storage. PersistentVolumes thus have a direct dependency on the storage of a specific Kubernetes cluster. Therefore, they cannot be used in another cluster without further considerations. Similarly, they can not be reused for separate installations of an application, for example for different customers or for different test environments, since the various instances are usually not supposed to use the same data.

In practice, PersistentVolumes are therefore rarely created directly. Instead, they are requested with a PersistentVolumeClaim. PersistentVolumeClaims decouple the management of storage from the specific storage architecture of a cluster. A PersistentVolumeClaim is a description of the desired properties of a volume. It does, however, not define where the volume's data actually resides. When a PersistentVolumeClaim is created in Kubernetes, Kubernetes tries to find an available PersistentVolume that meets the requirements of the PersistentVolumeClaim. If such a PersistentVolume exists, it is bound to the PersistentVolumeClaim. The relationship between PersistentVolumeClaim and PersistentVolume is bidirectional; a PersistentVolumeClaim must reside entirely on a single PersistentVolume and, in return, each PersistentVolume can host only one PersistentVolumeClaim.

Once a PersistentVolume is bound to a PersistentVolumeClaim, the PersistentVolumeClaim represents the ownership of that PersistentVolume. As long as the PersistentVolumeClaim is not deleted, the PersistentVolume remains existing. However, if a PersistentVolumeClaim is removed, depending on the retention policy of the PersistentVolume, the PersistentVolume and all data on it may also be deleted.

To access the data of a PersistentVolume, the PersistentVolumeClaim to which the PersistentVolume is bound is mounted as a volume in a Pod. This can be done in multiple Pods, but depending on the type of PersistentVolume, all Pods accessing the same PersistentVolume may then need to run on the same Kubernetes Node.

Most Kubernetes clusters contain a so-called storage provisioner and a StorageClass. The storage provisioner is software specific to the storage used. The StorageClass, however, is a configuration object. The StorageClass describes which provisioner should create new volumes and which parameters should be used by the provisioner. For example, a StorageClass can use a cloud provider's provisioner and specify as parameters the storage type to buy and the file system to use. The provi-

sioner then creates a PersistentVolume according to the specifications from the PersistentVolume-Claim and the StorageClass. This PersistentVolume is then bound to the PersistentVolumeClaim. If a PersistentVolumeClaim is created without any special information, the Default-StorageClass is responsible for this claim. Besides the default StorageClass, additional StorageClasses, which can then be referenced in PersistentVolumeClamais, may exist.

When installing applications on Kubernetes, PersistentVolumes are rarely created directly. Instead, a PersistentVolumeClaim is created and Kubernetes is trusted to bind the PersistentVolumeClaim to a PersistentVolume. This makes it possible to install an application with the same yaml files multiple times, for example in different Namespace, in different Kubernetes clusters or with different cloud providers, since in all cases a new PersistentVolume is created and bound to a PersistentVolume specific its environment.

5.2. Example: PersistentVolume (Rarely Created Manually)

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: training-webserver-htdocs
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
    - ReadWriteMany
    - ReadOnlyMany
  hostPath:
    path: /tmp/test
```

5.3. Example: PersistentVolumeClaim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: training-webserver-htdocs
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

5.4. Example: Pod with PVC Reference

```
apiVersion: v1
kind: Pod
metadata:
  name: training-webserver-pvc
  labels:
    app: training
    component: webserver
spec:
  containers:
    - name: webserver
      image: httpd
      volumeMounts:
      - mountPath: /usr/local/apache2/htdocs
        name: htdocs
  volumes:
    - name: htdocs
      persistentVolumeClaim:
        claimName: training-webserver-htdocs
```

6. Services

6.1. Explanations

A service is used to address processes running in Pods via network. Even though these processes are also accessible via the IP addresses of the Pods, the IP addresses of the Pods can change during an update. Furthermore, it happens frequently that multiple similar Pods run and that requests should be distributed across all these Pods.

Services can have different types, which are mainly derived from the basic type `ClusterIP`. For a ClusterIP Service, Kubernetes reserves an IP address from a special address range and adds this IP address to the DNS under the name of the Service. Other Pods can now resolve the IP address using the name of the Service via DNS. The definition of a Service also contains the ports under which the Service should be accessible. For each port of the Service, the destination port in the respective Pods can also be specified. For example, a Service can receive HTTP connections on port 80 and forward them to port 8080 in the Pods.

To which Pods a Service forwards requests is determined by a label selector in the Service. All Pods whose labels match this selector will be considered by the Service. Therefore it is important to define labels in such a way that exactly those Pods can be found that offer the same service and can answer requests in the same way.

Services with this basic functionality can be configured in three variants:

- `ClusterIP` reserves a separate IP address for the service, adds it to the DNS under the name of the Service, and forwards incoming requests to one of the Pods matched by the Service selector.
- `NodePort` works like ClusterIP, except that in addition, for each port of the Service, an additional port is selected from a specially defined range. This port can then be used to access the Service under the IP addresses of the Kubernetes Nodes.
- `Loadbalancer` works like ClusterIP, except that Kubernetes additionally requests a load balancer with a public IP address for the Service. This requires the availability of a load balancer, which a lot of cloud providers offer as part of their managed Kubernetes and which has to be additionally configured for custom Kubernetes setups.

6.2. Example: Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: training-webserver
spec:
  ports:
    - port: 80
      protocol: TCP
      targetPort: 80
  selector:
    app: training
    component: webserver
  type: NodePort
```

7. Startup, Readiness and Lifeness Probes

7.1. Explanations

If the Pod is matched by the selector of a Service, the Pod receives requests through the service as long as it is in the ready state. Without further configuration, this is the case when that Pod started and did not terminate. However, the start of a container does not mean that the container is actually able to answer requests. It may take some time for the application in the container to initialize. In addition, situations may occur where the application is overloaded, or where an error has occurred that does not cause the application to crash, but still results in incorrect responses.

Kubernetes supports three tests to examine the state of a container in more detail:

- The `readinessProbe` is executed continuously. It decides whether a container is considered ready or not. This can be seen, for example, in the output of `kubectl get pods`. The ready state is important for Services. Only Pods that are ready receive requests from a Service. Furthermore, various orchestration objects take the ready status in consideration during rolling updates. If a container does not have a readiness probe, it is always considered ready.
- The `livenessProbe` is executed continuously. If it fails, the Pod is restarted.
- The `startupProbe` is executed when the Pod is started. Only if the startup probe is successful, the readiness probe and the liveness probe for the respective container are started. The startup probe is then not executed any further. If the startup probe fails permanently, the Pod is restarted.

All three probes are performed by the kubelet responsible for the Pod. Probes are defined individually for each container. There are four types of tests supported by all three probes:

- `exec` executes a command in the container and expects a return value `0`.
- `httpGet` retrieves a path from the Pod's IP address via HTTP and expects an HTTP response code that signals successful processing.
- `tcpSocket` establishes a TCP connection to a port on the IP address of the Pod and expects the connection to be accepted.
- `grpc` establishes a GRPC connection to a port on the IP address of the Pod and expects the connection to be accepted.

By default, a probe is considered to be passed if it has worked once. It is considered failed, by default when it has failed three consecutive times. The number of required attempts, the interval of the probes as well as an initial delay for the start of the probes can be adjusted.

7.2. Example: Pod with Startup, Readiness and Liveness Probes

```
apiVersion: v1
kind: Pod
metadata:
  name: training-webserver-probes
spec:
  containers:
    - name: webserver
      image: httpd:latest
      readinessProbe:
        httpGet:
          path: /ready.txt
          port: 80
      livenessProbe:
        tcpSocket:
          port: 80
      startupProbe:
        exec:
          command:
            - cat
            - /tmp/started.txt
```

8. Manually Defined Endpoints

8.1. Explanations

For all Services with a selector, Kubernetes automatically creates an Endpoint object whose name is identical to the name of the Service. In the Endpoint object, Kubernetes maintains a list of target addresses whose Pods are in ready state and another list of target addresses whose Pods are not in ready state. By offloading the Pod IPs identified by the Service's selector to a separate Endpoint object automatically managed by Kubernetes, the Service object can be changed without disrupting the function of the Service.

In principle, Services can also be created without a selector. They will then still receive an IP address and a DNS entry. However, no Endpoint object is created, which results in the Service having no targets. If an Endpoint object with the Service's name is created manually, the addresses specified in it appear as targets in the Service of the same name. This makes it possible to make services that do not run within Kubernetes accessible via a Service. For example, this can provide a consistent hostname for the database server, even if the database server is a traditionally administered database cluster outside of Kubernetes.

8.2. Example: Service without Selector

```
apiVersion: v1
kind: Service
metadata:
  name: training-db-manual
spec:
  type: ClusterIP
  ports:
    - port: 3306
      protocol: TCP
```

8.3. Example: Endpoints

```
apiVersion: v1
kind: Endpoints
metadata:
  name: training-db-manual
subsets:
  - addresses:
      - ip: 192.168.122.90
    ports:
      - port: 3306
        protocol: TCP
```

9. ExternalName Service

9.1. Explanations

Services of the type `ExternalName` neither have a selector nor a specification of ports. Instead, they only have the `externalName` attribute, which points to another DNS name. The Service is then added to the DNS as a `CNAME` record, which points to the external DNS name.

This allows external targets whose IP address may change to be made available in Kubernetes. For example, either the productive API or the testing API of a payment service provider can be offered under a standardized DNS name this way.

When connecting to an ExternalName Service, it is important to keep in mind that the TLS certificates used by the destination are probably not valid for the name of the Service. If necessary, the application that establishes the connection should first determine the actual destination hostname by resolving the CNAME record, and then construct the final URL from it.

9.2. Example: ExternalName Service

```
apiVersion: v1
kind: Service
metadata:
  name: xamira
spec:
  type: ExternalName
  externalName: www.xamira.de
```

10. Headless Service

10.1. Explanations

A headless service is a ClusterIP Service without a cluster IP. There is therefore no network endpoint at which the Service itself could be reached, hence the term "headless". However, a headless Service is still added to the DNS. Instead of a cluster IP, all IP addresses of the Endpoints found by the Service are added directly in the DNS as individual records. A name lookup then returns the IP addresses of all Pods found by the service, if the Pods are ready.

Using a headless service, applications can find out whether a Service actually has Endpoints and, if so, what they are. This information is especially important when multiple Pods should form a cluster and need information about the other cluster members to do so. This information would not be available via a normal Service, since a normal Service publishes a cluster IP in the DNS even if it has no Endpoints, and on the other hand, even if there are Endpoints, it is not transparent in DNS which Endpoints there are.

10.2. Example: Headless Service

```
apiVersion: v1
kind: Service
metadata:
  name: training-webserver-headless
spec:
  type: ClusterIP
  clusterIP: None
  selector:
    app: training
    component: webserver
  ports:
    - port: 80
      protocol: TCP
```

11. Ingress

11.1. Explanations

An Ingress is a reverse proxy that receives external requests and forwards them to Kubernetes Services according to a set of rules. For HTTP requests, the host header and the requested path are taken into account to select the appropriate target Service.

The actual proxy server functionality is provided by an Ingress controller. However, it is not part of Kubernetes and must either be installed or, in the case of a managed Kubernetes, is provided by the respective cloud provider. Depending on which Ingress controller is used, it may be possible to specify additional options in the form of annotations in the Ingress resource.

11.2. Example: Ingress

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: training-webserver
spec:
  ingressClassName: nginx
  rules:
    - host: www1.10.0.0.1.nip.io
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: training-webserver
                port:
                  number: 80
```

11.3. Example: Ingress with Rewrite

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: training-webserver-paths
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /$2
spec:
  ingressClassName: nginx
  rules:
    - host: www2.10.0.0.1.nip.io
      http:
        paths:
          - path: /www(/|$)(.*)
            pathType: Prefix
            backend:
              service:
                name: training-webserver
                port:
                  number: 80
```

12. ConfigMaps

12.1. Explanations

ConfigMaps contain key-value pairs. Pods can be configured to make the values available in the containers of the Pod. Thus, values specific to a concrete installation of an application can be defined in a ConfigMap and then referenced from within the Pods.

In a Pod, a ConfigMap can be referenced in two ways. On the one hand, the values of ConfigMaps can be mapped to environment variables of a container. On the other hand, ConfigMaps can be mounted as volumes. In this case, each key appears as a file containing the respective value. The `|` operator can be used to specify multi-line values in yaml, but care must be taken to indent all lines correctly.

With a ConfigMap, all installation-specific values can be excluded from the Pods. The Pod definitions are then universally valid and can be used identically for many instances, for example for different test systems or different customers. In addition, ConfigMaps avoid repetition: For example, if the name of a database is stored once in a ConfigMap, multiple Pods can use the ConfigMap to access the name. This ensures, for example, that a container performing a database migration migrates the same database that is accessed by the application server.

12.2. Example: ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: training-shell-config
data:
  application_name: Super Training App
  mysetup.ini: |
    [general]
    type=production
```

12.3. Example: Pod with ConfigMap Reference

```
apiVersion: v1
kind: Pod
metadata:
  name: training-shell-config
spec:
  containers:
    - name: containername
      image: alpine:latest
      stdin: true
      tty: true
      env:
        - name: APPLICATION_NAME
          valueFrom:
            configMapKeyRef:
              name: training-shell-config
              key: application_name
      volumeMounts:
        - name: cmvol
          mountPath: /etc/config
  volumes:
    - name: cmvol
      configMap:
        name: training-shell-config
```

13. Secrets

13.1. Explanations

Secrets, like ConfigMaps, contain key-value pairs that can be used as environment variables or files in a Pod.

While ConfigMaps are intended for general configuration data, the purpose of Secrets is to store confidential information. Kubernetes can be configured to store Secrets in encrypted form — although this is not the case by default. This protection only means that the data is stored encrypted on the Kubernetes Nodes. Given sufficient permission, the data is still accessible unencrypted via the API. In the yaml definition of a secret, the values are specified base64 encoded, but this is only encoding and not encryption.

13.2. Example: Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: training-shell-secret
data:
  password: dG9wc2VjcmV0
```

13.3. Example: Pod with Secret Reference

```
apiVersion: v1
kind: Pod
metadata:
  name: training-shell-secret
spec:
  containers:
    - name: containername
      image: alpine:latest
      stdin: true
      tty: true
      env:
        - name: PASSWORD
          valueFrom:
            secretKeyRef:
              name: training-shell-secret
              key: password
      volumeMounts:
        - mountPath: /etc/secret
          name: secretvol
  volumes:
    - name: secretvol
      secret:
        secretName: training-shell-secret
```

14. ReplicaSet

14.1. Explanations

A ReplicaSet ensures that a certain number of similar Pods are running. For this purpose, the ReplicaSet contains a selector that matches the Pods for which it is responsible. If the number of Pods found does not match the specified number of replicas, the ReplicaSet creates new Pods according to the template specified in the ReplicaSet. By adjusting the number of desired replicas, the number of Pods can be scaled.

Besides the ability to change the number of Pods, ReplicaSets ensure that Pods are restarted after a Node failure. Therefore, Pods are usually not created directly, but are always created by an orchestration object such as a ReplicaSet, even if the number of replicas is only 1.

Even though the template of a ReplicaSet can be modified, this does not immediately lead to changes to the running Pods. Only when new Pods need to be created, these new Pods are created according to the new template. This is dangerous because Pods with different definitions can be active in parallel in the same ReplicaSet, and errors in the template may remain undiscovered. Therefore, the template of a ReplicaSet is usually not changed. Instead, Deployments, that are made specifically to manage changes in the template, are used in practice. ReplicaSets are however still important, because Deployments internally leverage ReplicaSets to control the individual Pods.

14.2. Example: ReplicaSet

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: training-webserver-rs
spec:
  replicas: 2
  selector:
    matchLabels:
      app: training
      component: webserver
  template:
    metadata:
      labels:
        app: training
        component: webserver
    spec:
      containers:
        - name: webserver
          image: httpd:latest
```

15. Deployment

15.1. Explanations

The properties of a Deployment are similar to those of a ReplicaSet. Deployments, however, differ from ReplicaSets in that they can handle changes in the template properly. For this purpose, Deployments do not start Pods directly, but instead create ReplicaSets. Deployments extend these ReplicaSets with an additional label in the template and in the selector, which contains a checksum of the original template. Now, if the template is changed in the Deployment, the Deployment creates a new ReplicaSet that, because of the additional label, creates Pods that fall only in the responsibility of the new ReplicaSet, but not in the responsibility of the old one. Now the Deployment can scale the new ReplicaSet up and scale the old ReplicaSet down. In doing so, Kubernetes takes care to keep a specific minimum number of Pods in the ready state to maintain the availability of the application provided by the Pods.

15.2. Example: Deployment

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: training-webserver-deploy
  annotations:
    kubernetes.io/change-cause: "Beschreibung der Änderung"
spec:
  replicas: 2
  selector:
    matchLabels:
      app: training
      component: webserver
  template:
    metadata:
      labels:
        app: training
        component: webserver
    spec:
      containers:
        - name: webserver
          image: httpd:latest
  strategy:
    rollingUpdate:
      maxUnavailable: 25%
      maxSurge: 25%
```

16. DaemonSet

16.1. Explanations

A DaemonSet starts exactly one replica of the Pod on each Node on which a Pod could in principle run. Again, a selector is responsible for identifying the Pods that belong to the DaemonSet and a template serves as a blueprint for missing Pods.

On which Nodes a Pod can run can be influenced by various options. These include the following options:

- The `nodeName` parameter can be used to specify the name of a node on which the Pod should run.
- A `nodeSelector` can be used to specify a selector that must be matched by the labels of the nodes that Kubernetes will consider for scheduling the Pod.
- If a Node is provided with a so-called taint, Pods are, depending on the type of taint, no longer executed on this Node or no longer assigned to this Node. A taint consists of a freely definable key, an optional value, and an effect that describes how the taint works. For example, the `NoSchedule` effect causes the node to stop receiving new Pods. However, with `tolerations`, a Pod can ignore taints, by specifying the key, the value if any, and the effect of each tolerable taint.

These possibilities exist in every Pod, no matter how it was started.

A DaemonSet attempts to launch one replica of the Pod on as many Nodes as possible. In doing so, the DaemonSet respects the Pod's constraints and adapts to change. For example, if a Node is labeled with a label that matches the Node selector of a Pod in a DaemonSet, the number of replicas in the DaemonSet automatically increases and the DaemonSet starts a new Pod on the freshly labeled Node.

DaemonSets are mainly used for infrastructure services, for example, monitoring agents or logging agents that should run on each Node by default.

16.2. Example: DaemonSet

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: training-webserver-ds
spec:
  selector:
    matchLabels:
      app: training
      component: webserver
  template:
    metadata:
      labels:
        app: training
        component: webserver
    spec:
      containers:
        - name: webserver
          image: httpd:latest
      nodeSelector:
        training: run-ds
      tolerations:
        - key: node-role.kubernetes.io/control-plane
          effect: NoSchedule
```

17. StatefulSet

17.1. Explanations

A StatefulSet creates a certain number of similar Pods. Different to a ReplicaSet, these Pods have special properties:

- The Pods are numbered sequentially, starting with the appendix `-0`.
- A Pod is not started until all Pods with a lower index have become ready. When a StatefulSet is deleted, one Pod at a time is deleted starting with the highest index, and one Pod must be fully terminated for the next Pod to be terminated.
- If the StatefulSet contains a `volumeClaimTemplate`, a separate PersistentVolumeClaim is created for each Pod. If a StatefulSet is deleted or scaled down, the PersistentVolumeClaims and thus also the PersistentVolumes bound to the PersistentVolumeClaim along with their data continue to exist. If the StatefulSet is later scaled up again, the new Pods get back the volumes for the respective instance.

Each StatefulSet has a Headless Service that is referenced in the StatefulSet. This service can be used to retrieve a list of all Pods that belong to the StatefulSet. Furthermore, the hostnames of individual Pods can be retrieved below the DNS name of the Service to specifically determine the IP address of a particular Pod.

StatefulSets are mainly used for applications whose instances form a cluster. For example, it is possible to tell from the host name of a Pod how many Pods already exist in the StatefulSet. In addition, individual instances of the StatefulSet can be addressed via the Service. In the instance-specific PersistentVolumes, for example, access keys to the cluster can be stored, with which the respective instance can regain its identity after a rescaling. In addition, data can be stored in the PersistentVolumes, which after upscaling no longer need to be completely transferred again but instead only needs to be updated by means of delta replication.

17.2. Example: Headless Service for StatefulSet

```
apiVersion: v1
kind: Service
metadata:
  name: training-stateful-web
spec:
  type: ClusterIP
  clusterIP: None
  ports:
  - port: 80
  selector:
    app: training
    component: stateful-web
```

17.3. Example: StatefulSet

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: training-stateful-web
  labels:
    app: training
    component: stateful-web
spec:
  replicas: 3
  serviceName: training-stateful-web
  selector:
    matchLabels:
      app: training
      component: stateful-web
  template:
    metadata:
      labels:
        app: training
        component: stateful-web
    spec:
      containers:
        - name: webserver
```

```
        image: nginx
        ports:
          - containerPort: 80
            name: web
        readinessProbe:
          httpGet:
            path: /ready.txt
            port: 80
        volumeMounts:
          - name: htdocs
            mountPath: /usr/share/nginx/html
volumeClaimTemplates:
  - metadata:
      name: htdocs
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
```

18. Job

18.1. Explanations

A Job starts a Pod once and ensures that the Pod exits again. Accordingly, the Pod's restart policy must be explicitly set to a value other than the default value `always`. When the Pod has terminated, it remains in the `completed` state until the Pod or Job is deleted.

Typical use cases for Jobs are initialization tasks during the installation or update of a software, for example loading sample data into a database, or performing database migrations.

18.2. Beispiel: Job

```yaml
apiVersion: batch/v1
kind: Job
metadata:
  name: training-count-once
spec:
  template:
    spec:
      containers:
        - name: training
          image: alpine:latest
          command:
            - "/bin/sh"
            - "-c"
            - "for i in $(seq 10); do echo $(date): $i; sleep 1; done"
      restartPolicy: Never
```

19. CronJob

19.1. Explanations

CronJobs execute processes at specific times. To do so, a CronJob starts a Job at each relevant point in time, which then in turn starts a Pod.

The point in time when a Job should run is determined by the `schedule` parameter. The value contains the following specifications separated by spaces:

- Minute as a number between `0` and `59`
- Hour as a number between `0` and `23`
- Day of the month as a number between `1` and `31`
- Month as a number between `1` and `12` or as a month in the form of `jan`, `feb`, `mar`, ...
- Day of the week as a number between `0` and `7`, where both `0` and `7` represent Sunday, or as an indication of the day in the form `mon`, `tue`, `wed`, ...

For each of these specifications, values can be defined in various forms:

- The special character `*` stands for any allowed value to execute a Job, for example, every minute (`* * * * *`).
- A single value such as `0` to run a Job, for example, always on the top of an hour (`0 * * * *`) or daily at a specific time such as 3:45 (`45 3 * * *`).
- A range of values like `0-4` to run a Job, for example, only at night (`* 0-4 * * *`).
- A specification like `*/5`, which is always true if the value is divisible by the specified number without rest. With the minute specification `*/5` a Job is for example executed 0, 5, 10, 15, 20, ... minutes after a full hour. It should be noted that only the actual value of the number matters. With the minute specification `*/11` a Job is executed at minute 55 of an hour and then 16 minutes later at minute 11 of the following hour.
- Multiple entries can be separated by commas, such as `15,45 22-23,0-4 * *`.

CronJobs can be used, for example, to automatically start administrative activities such as creating backups, cleaning up caches, or starting billing cycles at specific times.

By default, Kubernetes keeps the last three successful Jobs and deletes the oldest Job when a new Job completes.

19.2. Example: CronJob

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: training-count-every-five-minutes
spec:
  schedule: "*/5 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: training
              image: alpine:latest
              command:
                - "/bin/sh"
                - "-c"
                - "for i in $(seq 10); do echo $(date): $i; sleep 1; done"
          restartPolicy: OnFailure
```

20. Resource Limits in Pods

20.1. Explanations

By default, Kubernetes does not limit the resource consumption of containers. Each container can consume all available resources. On the other hand, Kubernetes also does not reserve resources for specific containers, so it may happen that Pods run together on a Node that is foreseeably not able to meet the requirements of the Pods.

To avoid such scenarios, resource consumption can be configured for each container:

- Limits specify the maximum amount of resources the container may use. Even if more resources are available, the consumption of the container is limited at the values specified here. Limits can be overcommitted, the sum of the limits of all containers may exceed the resources available on the Node.
- Requests specify with which resource consumption Kubernetes should plan for the container. A new Pod is started on a Node only if there are enough resources available to host the new Pod, taking into account the resource requests of all other containers running on the Node. An overcommitment is not possible for requests, if the requested resources are not free on any Node, the Pod remains in the pending state and is not executed.

The two most important resource limits concern CPU and memory:

- `cpu` specifies the CPU consumption in logical CPUs, where the unit `m` stands for milli CPUs, so `500m` is identical to `0.5` CPUs.
- `memory` specifies the memory consumption. Here, both base 2 units (`Ki`, `Mi`, `Gi`, ...) as well as base 10 units (`k`, `M`, `G`, ...) can be used.

In practice, the limits for containers should be set generously. They primarily serve as a safety net when applications allocate unnecessary resources in the event of an error. Resources, on the other hand, should be allocated with caution, as they may block resources and cause Pods to fail to be scheduled to a Node and execute. In any case, the actual resource consumption should be permanently monitored and requests and limits should be adjusted when necessary.

Availability scenarios must be considered when sizing Pods. When a Node fails, the orchestration objects restart the Pods previously located there, causing those Pods to be distributed to other Nodes. This assumes that there are Nodes with enough free resources to host the Pods. Therefore, it may be beneficial to create more small Pods rather than a few large ones — the likelihood of multiple small Pods being assigned to multiple Nodes is higher than the chance that a very large Pod can be assigned to a single, sufficiently free Node.

Some applications adapt their behavior to the memory available to them. Such applications should be explicitly configured for the resource requests and limits of their containers. A prominent example is the Java VM, for which numerous references to the corresponding configuration options can be found on the Internet.

20.2. Example: Pod with Resource Configuration

```
apiVersion: v1
kind: Pod
metadata:
  name: training-shell-limited
spec:
  containers:
    - name: containername
      image: alpine:latest
      stdin: true
      tty: true
      resources:
        requests:
          memory: 128Mi
          cpu: 500m
        limits:
          memory: 256Mi
          cpu: "1"
```

21. HorizontalPodAutoscaler

21.1. Explanations

HoritzontalPodAutoscaler observe the utilization of the Pods that a scalable object, usually a Deployment, creates. If the utilization of the Pods exceeds a defined threshold, the number of replicas of the scalable object is adjusted to reduce the overall utilization below the specified threshold. This is based on the assumption that additional replicas reduce the utilization of all replicas, which is usually the case if a Service continuously distributes incoming requests across all replicas.

HorizontalPodAutoscalers rely on metrics regarding the Pods, therefore a metrics server must be installed in Kubernetes to provide this data. The reference for the percentages for metrics of type `Resource` are the requests and limits of the Pods, which is why a HoritzontalPodAutoscaler can only use these metrics for Pods that have a resource configuration.

21.2. Example: HorizontalPodAutoscaler

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: training-webserver-hpa
spec:
  minReplicas: 1
  maxReplicas: 10
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: training-webserver
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

22. VerticalPodAutoscaler

22.1. Explanations

VerticalPodAutoscalers attempt to respond to load changes by adjusting the requests and limits of Pods. Because these values cannot be changed for running Pods, VerticalPodAutoscaler delete Pods that do not match the current VerticalPodAutoscaler recommendations. The Pods are then recreated by an orchestration object, such as a ReplicaSet. The new Pods pass through the Kubernetes API, where an admission controller, which is also part of the VerticalPodAutoscaler, modifies the resource configuration.

Currently, VerticalPodAutoscaler are not yet part of Kubernetes and need to be installed later if required.

22.2. Example: VerticalPodAutoscaler

```yaml
apiVersion: autoscaling.k8s.io/v1
kind: VerticalPodAutoscaler
metadata:
  name: training-webserver-vpa
spec:
  targetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: training-webserver
  updatePolicy:
    updateMode: Recreate
  resourcePolicy:
    containerPolicies:
      - containerName: '*'
        minAllowed:
          cpu: 250m
          memory: 128Mi
        maxAllowed:
          cpu: 1
          memory: 512Mi
        controlledResources:
          - cpu
          - memory
```

23. LimitRange

23.1. Explanations

By default, containers can be created in Kubernetes with either no resource specifications at all or with resource specifications of any size. This behavior can be changed by creating a LimitRange object in a Namespace. Henceforth, all new Pods created in the Namespace must conform to the LimitRange requirements.

Limits can be defined for both individual containers as well as for the sum of the containers within a Pod. The minimum and maximum limits of a single container or the sum of the limits of all containers in a Pod are specified with `min` and `max` values, respectively. Larger or smaller Pods can then no longer be created.

For containers, default values which always apply if no specifications have been made for a container can be specified for limits (parameter `default`) and requests (parameter `defaultRequest`). Since resources are always defined for individual containers, the default values can only be specified at container level but not for entire Pods.

23.2. Example: LimitRange

```
apiVersion: v1
kind: LimitRange
metadata:
  name: training-namespace-low-limits
spec:
  limits:
    - type: Container
      defaultRequest:
        cpu: 100m
        memory: 64Mi
      default:
        cpu: 200m
        memory: 128Mi
      min:
        cpu: 100m
        memory: 16Mi
      max:
        cpu: 250m
        memory: 256Mi
    - type: Pod
      min:
        cpu: 100m
        memory: 16Mi
      max:
        cpu: "0.5"
        memory: 512Mi
```

24. ResourceQuota

24.1. Explanations

ResourceQuotas define for an entire Namespace how many resources can be claimed in total in the resource requests and limits of all Pods running there. In addition, ResourceQuota can limit the number of certain objects. ResourceQuotas always apply to the Namespace in which the object resides.

24.2. Example: ResourceQuota

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: training-namespace-small-quota
spec:
  hard:
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi
    count/pods: "4"
    count/persistentvolumeclaims: "2"
    count/services: "2"
    count/services.nodeports: "1"
```

25. NetworkPolicy

25.1. Explanations

NetworkPolicies limit network traffic between Pods. Within a NetworkPolicy, a pod selector specifies to which Pods the NetworkPolicy applies. Once a policy applies to a Pod, that Pod is excluded from all network communication in the direction of the policy, unless a NetworkPolicy explicitly allows the communication.

A NetworkPolicy limits incoming, outgoing, or both incoming and outgoing traffic from the perspective of the Pods selected with the policy's Pod selector. In order to allow traffic, ingress or egress rules are defined, whereby ingress has nothing to do with the Ingress object for reverse proxies.

Within each rule, connections are defined by specifying ports and/or communication partners. Communication partners can either be Pods or IP addresses. Pods are specified by a Pod selector. To authorize Pods in other Namespaces, an additional Namespace selector can be specified. Since Namespaces contain a label with their name by default, cross-namespace communication can be easily allowed.

Namespaces are often equipped with a default NetworkPolicy. It has an empty Pod selector and thus applies to all Pods in the Namespace. This initially denies all network communication. At the same time, such default policies often allow base services, such as DNS. Application-specific communication must then explicitly be permitted by additional policies. Alternatively, communication with all Pods in the Namespace can be allowed in order to only deny communication with other Namespaces.

25.2. Example: NetworkPolicy

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: npol-training-webserver
spec:
  podSelector:
    matchLabels:
      app: training
      component: webserver
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              project: training
        - podSelector:
            matchLabels:
              app: training
      ports:
        - protocol: TCP
          port: 80
  egress:
    - to:
        - ipBlock:
            cidr: 192.168.99.0/24
            except:
              - 192.168.99.5/32
      ports:
        - protocol: TCP
          port: 80
```

26. Roles and RoleBindings

26.1. Explanations

By default, Kubernetes uses Role Based Access Control, or RBAC, to manage permissions in the cluster:

- `Roles` describe actions that may be performed on specific objects, such as listing Pods.
- `RoleBindings` define which users take a Role and thus have the permissions of the Role.

Multiple RoleBindings can refer to the same Role and the same user. Thus, several users can exercise the same Role and, on the other hand, one user can have multiple Roles. The permissions then add up accordingly.

Roles and RoleBindings each refer to the Namespace in which they reside. Similarly, there are also ClusterRoles and ClusterRolebindings that set permissions for the entire Kubernetes cluster.

26.2. Example: Role

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: role-training
rules:
  - apiGroups:
      - ""
    resources:
      - pods
    verbs:
      - get
      - watch
      - list
```

26.3. Example: RoleBinding

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: rolebinding-training
subjects:
  - apiGroup: rbac.authorization.k8s.io
    kind: User
    name: training-user
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: role-training
```