Module 3 Kubernetes

Orchestrating Containers

What We Will Cover

Introduction

Running Application: Resource Types

Accessing Applications: Services and Ingress

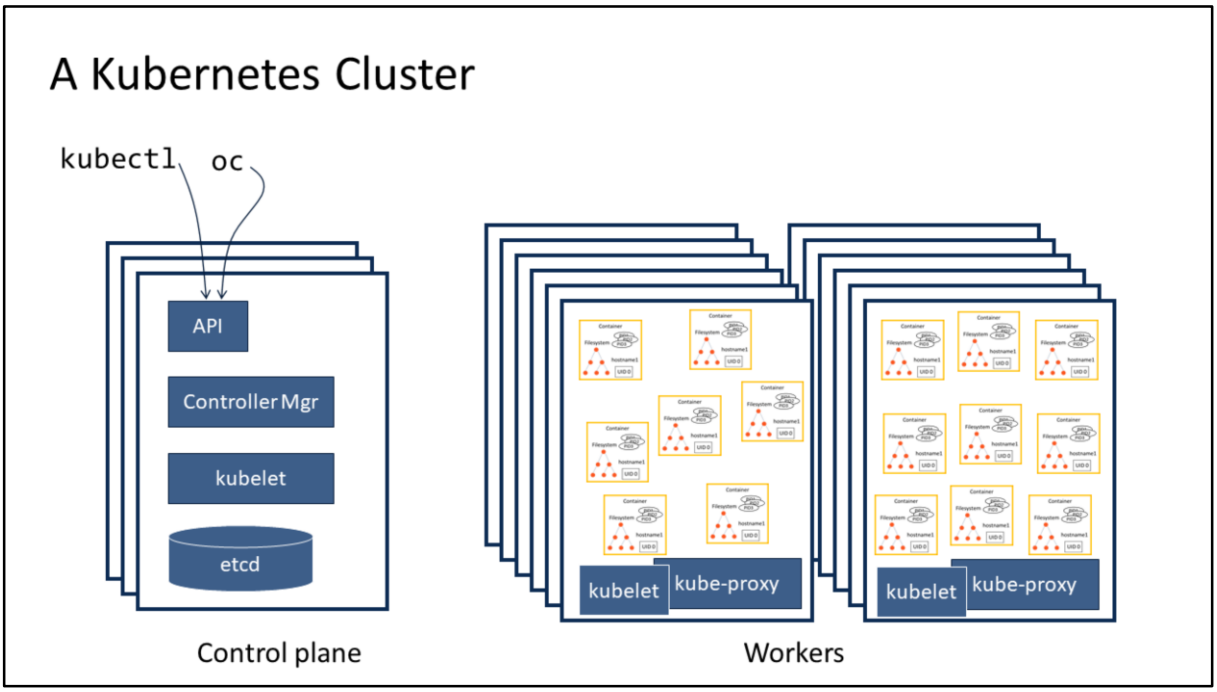
How to Handle Data: Volumes and Volume Claims

Configuring Applications: ConfigMaps and Secrets

Packaging Applications: Helm

Kubernetes

A quick introduction



An *application* consists of more than just a container. A web server, for example, may scale up or down automatically depending on its load. It needs a load balancer as a front and shared storage or a database for the site content. The original Docker has no tools to manage applications (Docker Swarm does).

The concepts that Kubernetes proposes *abstract from containers* and allow the management of applications. Kubernetes concepts address questions like what to do when a container crashes, how to scale applications up or down, how to upgrade application code with minimal or no service disruption, how to distribute applications over several container hosts, how to access distributed applications over the network.

Kubernetes manages clusters of container hosts. A Kubernetes cluster consists of control hosts and workers. Applications run on the workers in containers; a control component named **kubelet** manages them, and another component, the **kube-proxy**, facilitates network access. The control hosts run several control components including an **API server** and a **controller manager**. Applications including GUIs and the command line tool kubelet, as well as the OpenShift tool oc, access the cluster via the API server.

Since most control components run in containers as well, control nodes also need kubelets. The cluster's current state is stored in a distributed key-value store named **etcd**.

Applications are managed by pieces of code named **controllers**, which are themselves managed by the controller manager. A controller's task is to continuously watch the application's current state and compare it with the desired state in the key-value store. If the states differ, the controller adapts the current state accordingly. Users or automatic processes can modify desired states via API requests.

kubectl

kubectl	verb	object type	object
	get	pod	
	describe	replicaSet, rs	
	create	deployment, deploy	
	edit	daemonSet, ds	
	delete	configMap	
	apply	secret	
	explain		
	...	...	

Sept 2021 (c) Bernd Bausch 2021 5

The `kubectl` command facilitates command-line interaction with a Kubernetes cluster, and just like the `docker` and `docker-compose` commands, it can be used imperatively or declaratively. The usual way to manage Kubernetes objects is the declarative approach, using YAML files similar to `docker-compose`. For example, each resource type has its own YAML file structure, but also other objects like volumes or secrets. Such object descriptions are named **manifests** in Kubernetes.

`kubectl`'s syntax is simple: *kubectl verb object* plus options. Most verbs act on one or more objects of one type; for example, `kubectl get pod` lists all pods that are visible to the user, whereas `kubectl describe pod mypod` displays a detailed description of a pod named `mypod`. Some `kubectl` verbs are identical to `docker` verbs such as `run`, `exec`, and `logs`.

The **`apply`** verb is an exception; it only acts on manifests, which it uses to create new objects or modify existing ones. The **explain** verb provides succinct documentation of the various resources in Kubernetes.

Client Configuration

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: LS0...
    server: https://api.crc.testing:6443
  name: api-crc-testing:6443
- cluster:
    certificate-authority: <FILE>
  name: minikube
contexts:
- context:
    cluster: api-crc-testing:6443
    namespace: default
    user: kubeadmin
  name: crc-admin
```

```
- context:
    cluster: minikube
    namespace: default
    user: minikube
  name: minikube
current-context: minikube
...
users:
- name: developer
  user:
    token: sha256~jKB-QMX5P5dvaDF...
- name: kubeadmin
  user:
    token: sha256~EuCfSDowf1lW...
- name: minikube
  user:
    client-certificate: <FILE>
    client-key: <FILE>
```

Sept 2021 (c) Bernd Bausch 2021 6

The $HOME/.kube/config file contains information about your Kubernetes and OpenShift clusters, and the user accounts that you use to access them. It's a YAML file and lists clusters, contexts and users.

A **cluster** is described by its API endpoint (a URL) and an X.509 certificate authority that is used for authentication. In the example, the CRC cluster (OpenShift) includes the entire CA in the config file, whereas the Minikube cluster's CA is stored in a separate crt file.

The **users** section stores users' credentials. The example uses so-called *bearer tokens* to authenticate with the OpenShift cluster and an X.509 certificate for Minikube.

A **context** is essentially a combination of cluster and user account, plus the default Kubernetes *namespace* that will be used for the login session.

Using Manifests

mypod.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: mySite
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
  - ....
  volumes:
  - ....
```

```
kubectl create -f mypod.yaml
kubectl apply -f mypod.yaml
kubectl replace -f mypod.yaml
kubectl delete -f mypod.yaml
```

Sept 2021 (c) Bernd Bausch 2021 7

Kubernetes resources are usually managed using **manifests** written in YAML. The advantage of this approach: You have object declarations in files that you can manage like source code, in particular in a version control system. On the other hand, for quick tests imperative commands like kubectl run etc exist. This course focuses on manifests.

The manifest shown here describes a **pod**, a set of containers and volumes. Pods are the smallest units in Kubernetes; you can't manage containers directly.

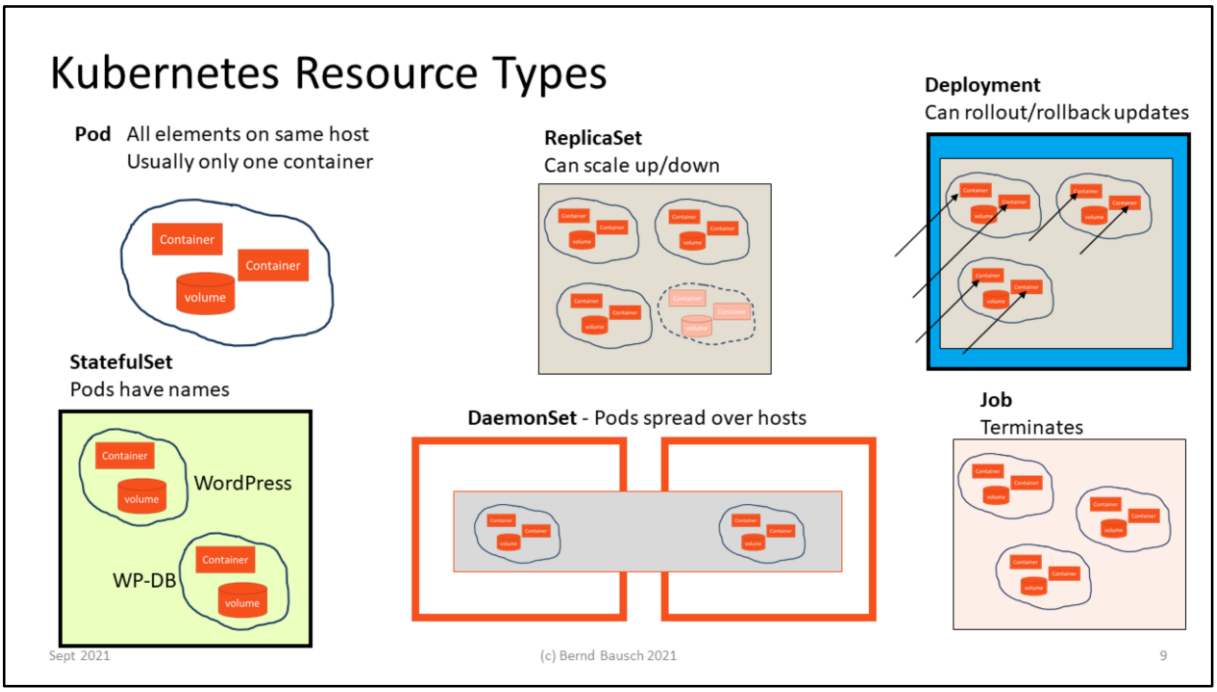
To create this pod, you have the option of `kubectl create` and `kubectl apply`. What is the difference? If a pod with the same name, website, exists already, `kubectl create` will fail. `kubectl apply` will compare the current status of website with the manifest and apply any differences.

`kubectl replace` only works if the website pod already exists. The difference between `apply` and `replace` is subtle. `apply` does not remove any properties of the object, but `replace` does. One could say that `apply` performs incremental updates without changing properties that are missing from the manifest, whereas `replace` removes all object properties and replaces them with the manifest description.

`kubectl delete` removes all objects that are described by this manifest. This is particularly useful when the manifest contains objects of several different types that, together, form an application. With one command, you can clean up all application resources.

Kubernetes Workload Types

Part 1: Pods, ReplicaSets, Deployments



This diagram summarizes the abstractions Kubernetes offers for managing applications. They are called *resource types*, or *workload types*. A controller is associated with each of them. They are documented at https://kubernetes.io/docs/concepts/workloads.

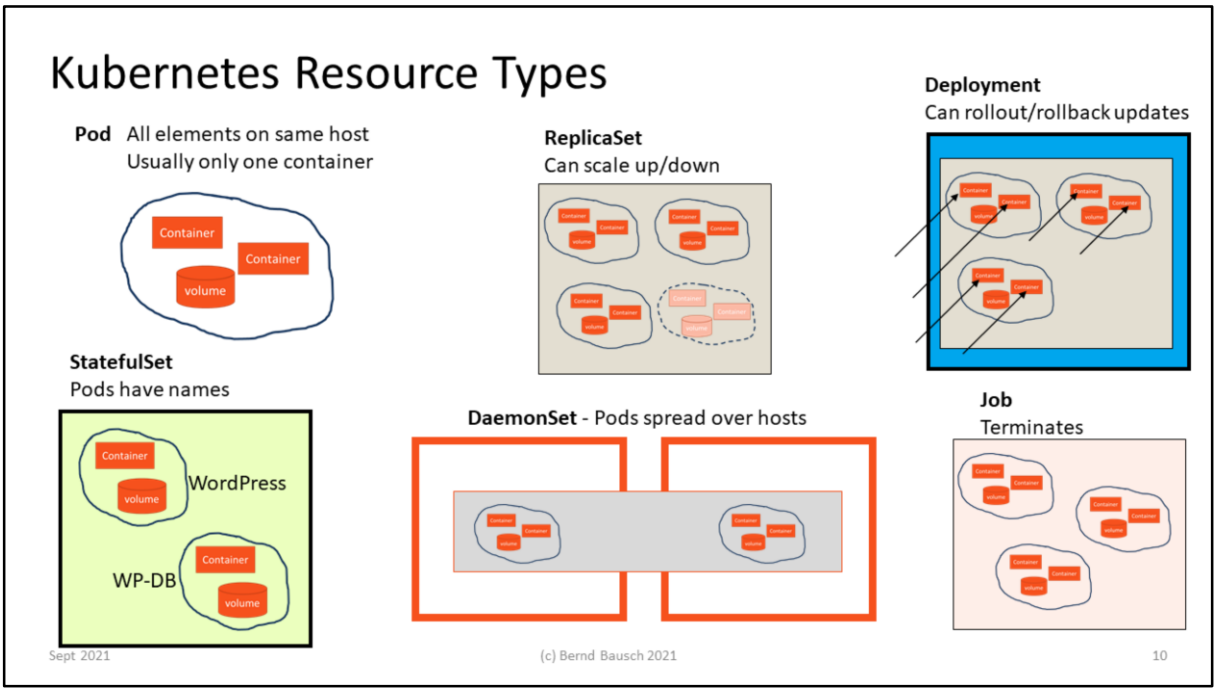
Kubernetes doesn't deal with containers alone. The smallest unit is a **pod**, a collection of containers and volumes that are located on the same container host. In practice, pods often have a single container, though another common pattern is a main container that processes or displays data, and a second container, often called "side-car", which might generate data or perform supporting tasks.

Running single pods in a Kubernetes cluster is possible, but not practiced very often. Instead, most pods are embedded in higher-level resource types, which all have specific purposes.

ReplicaSets are groups of identical pods that can be **scaled up or down** by changing the number of replicas. A typical example is a web site managed by several web server pods. When the load on the web site increases, more web server pods are added (scaling up); when it goes down, some pods are discarded.

Deployments, perhaps the most common way of implementing applications on Kubernetes, each include a single ReplicaSet. A deployment allows the pods in the ReplicaSet to be **upgraded** to more recent image versions. This is a rolling upgrade - not all pods are upgraded at the same time, but only a subset of all pods at a time, in order to avoid disrupting the application. Deployments can report the status of an upgrade, and allow **rolling it back** when it turns out that it introduces problems.

ReplicationControllers, not show on this diagram, are now deprecated precursors of Deployments. They combine the features of ReplicaSets and Deployments in one resource type.



Pods in a ReplicaSet are essentially anonymous. A pod's name and IP address are determined by Kubernetes and are not predictable. When a pod in a ReplicaSet crashes, the ReplicaSet controller launches a new pod with a different name and address.

This is not always desirable. For example, think of a WordPress application that requires a database pod. The WordPress server needs to know the DB pod's name or address. Should the database pod crash, and need to be replaced, the replacement has to have the same identity. A **StatefulSet** ensures that pods have the same identity, even when they are replaced.

Normally, it is not important on which host a pod runs. There are, however, applications that require pods on specific container hosts, or one pod on each container host. Typical cases are applications that manage the cluster's infrastructure, e.g. pods that monitor the hosts.

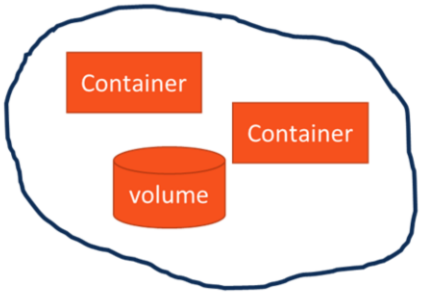
The **DaemonSet** was created to address such needs. Pods in a DaemonSet are guaranteed to run on all hosts or a subset of the hosts in the cluster.

All resource types discussed so far expect pods to run continuously, providing a 24/7 service. Think of web servers or monitoring pods. **Jobs** are collection of pods that perform a single task, then exit. A variation of Jobs are **CronJobs**, which perform tasks repeatedly based on a schedule.

Note that none of these resource types answer the question how applications can be accessed over the network. Another set of Kubernetes objects, so-called services, addresses this problem.

Pod

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: mySite
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
  - ....
  volumes:
  - ....
```




```
kubectl apply -f mypod.yaml
kubectl get pod
kubectl get pod mypod -o yaml
kubectl describe pod mypod
kubectl describe pod/mypod
kubectl delete pod/mypod
kubectl delete -f mypod.yaml
```

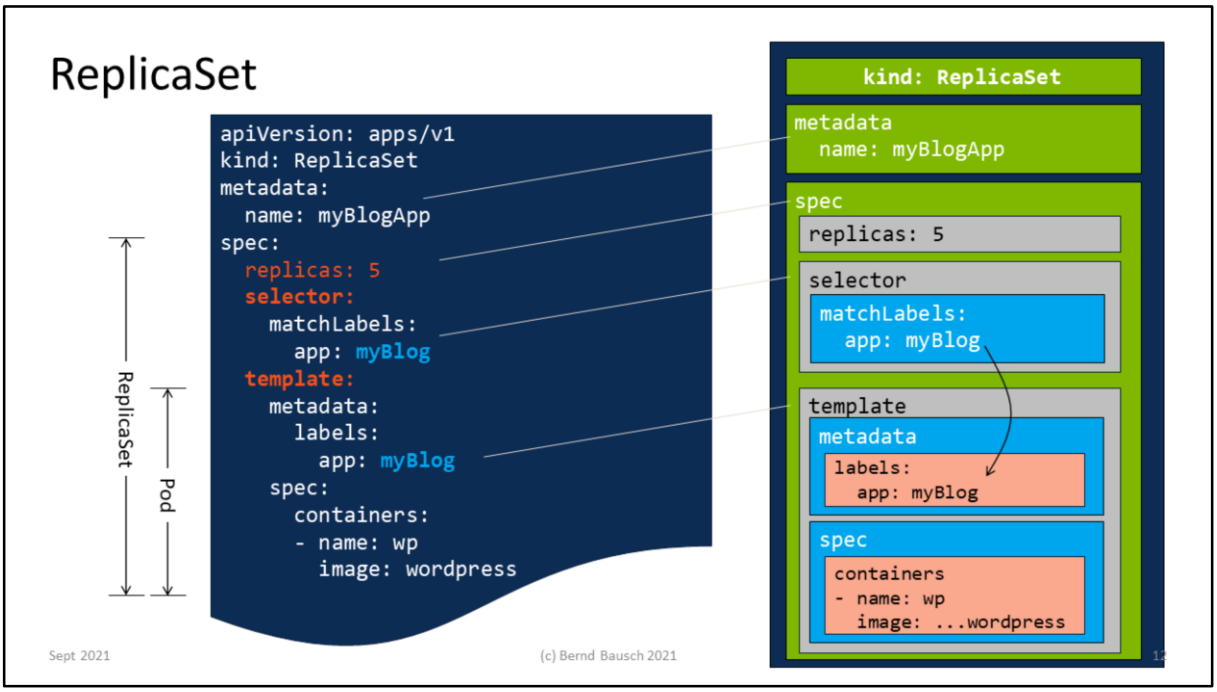
Sept 2021 (c) Bernd Bausch 2021 11

The declaration of a Kubernetes object always contains an API version, the object type (*kind*), metadata and a specification.

A pod's specification lists its containers and volumes together with their properties. Volumes in a pod are not persistent; i.e. they are deleted when the pod is deleted.

The slide shows a few common commands. They can be extended to all workload types; for example, the usual way to list all ReplicaSets is kubectl get replicaset or the abbreviated version kubectl get rs.

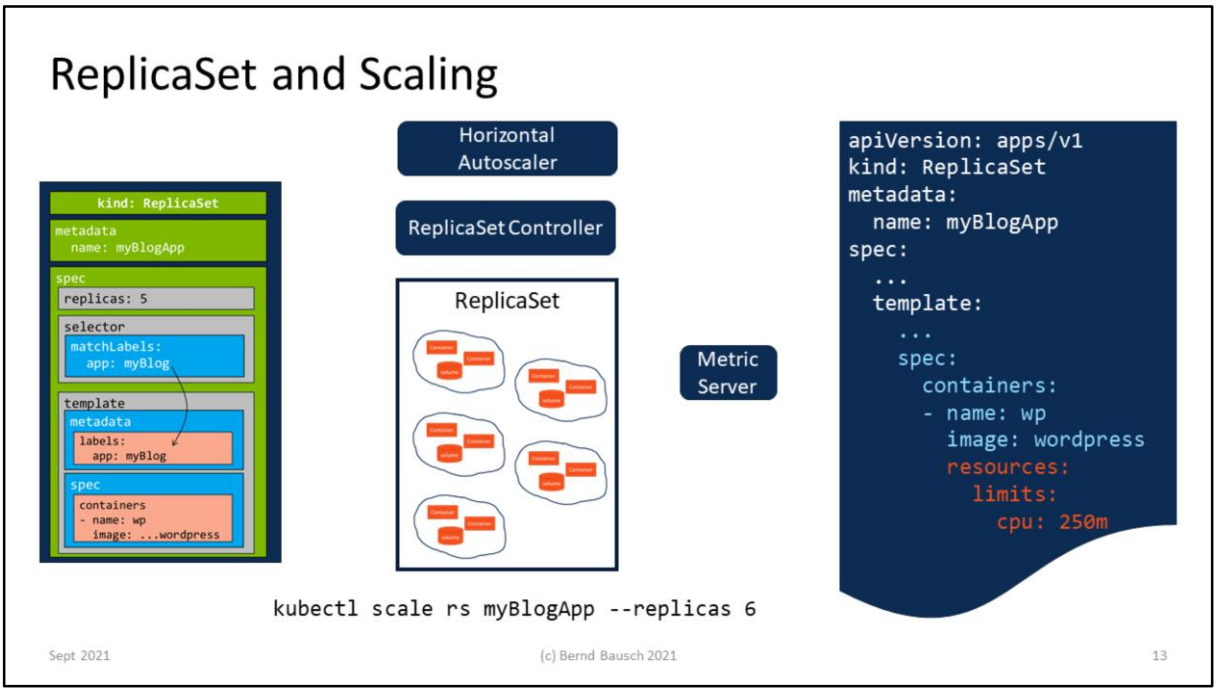
When a pod has a single container, you can enter that container with kubectl exec -it mypod bash, the same syntax as docker.



A ReplicaSet defines a set of identical pods. The ReplicaSet controller ensures that the number of pods is identical to the number of replicas in the ReplicaSet definition, i.e. the set's desired state. When a pod disappears, it creates a new pod. When the replica number is changed via an API, the controller removes or adds pods as needed.

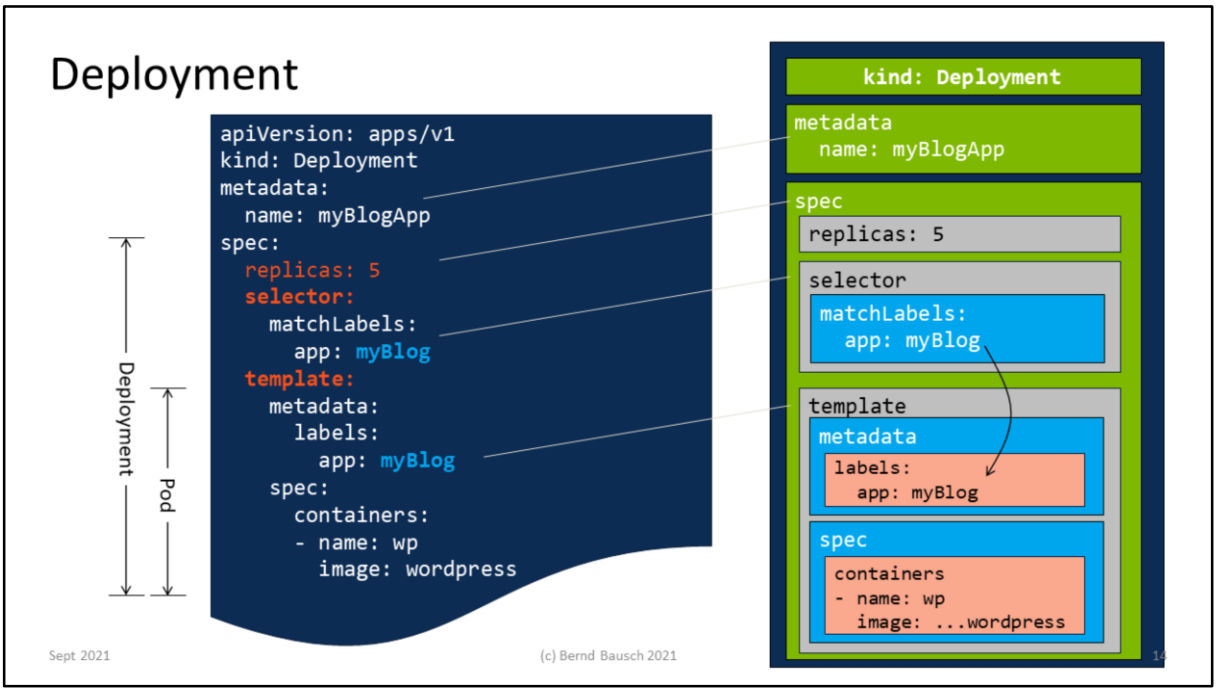
Therefore, a ReplicaSet's specification contains the description of the pods in the set under the `template` key. In addition, it includes the number of **replicas** and a **selector**, which defines exactly which pods are in the set. This selection is made with labels. A **label** is a key-value pair defined by the author of the ReplicaSet. In the example, all pods in the set have the label `app: myBlog`, where `app` it the key and `myBlog` is the value.

If the pod labels and the selector labels in a manifest don't match, ReplicaSet creation is rejected.



The purpose of ReplicaSets is scaling. You can **scale manually** using `kubectl scale` on the command line or by changing the `replicas` field in the ReplicaSet's manifest then reapplying it.

You can also configure the horizontal pod **autoscaler**. It will scale up and down until the ReplicaSet is close enough to a certain CPU usage target, which you define with the `kubectl autoscale` command or by applying a Horizontal Pod Autoscaler manifest. To enable autoscaling, the **metric server** must be deployed in the cluster, and you must configure **resource limits** in the pods.



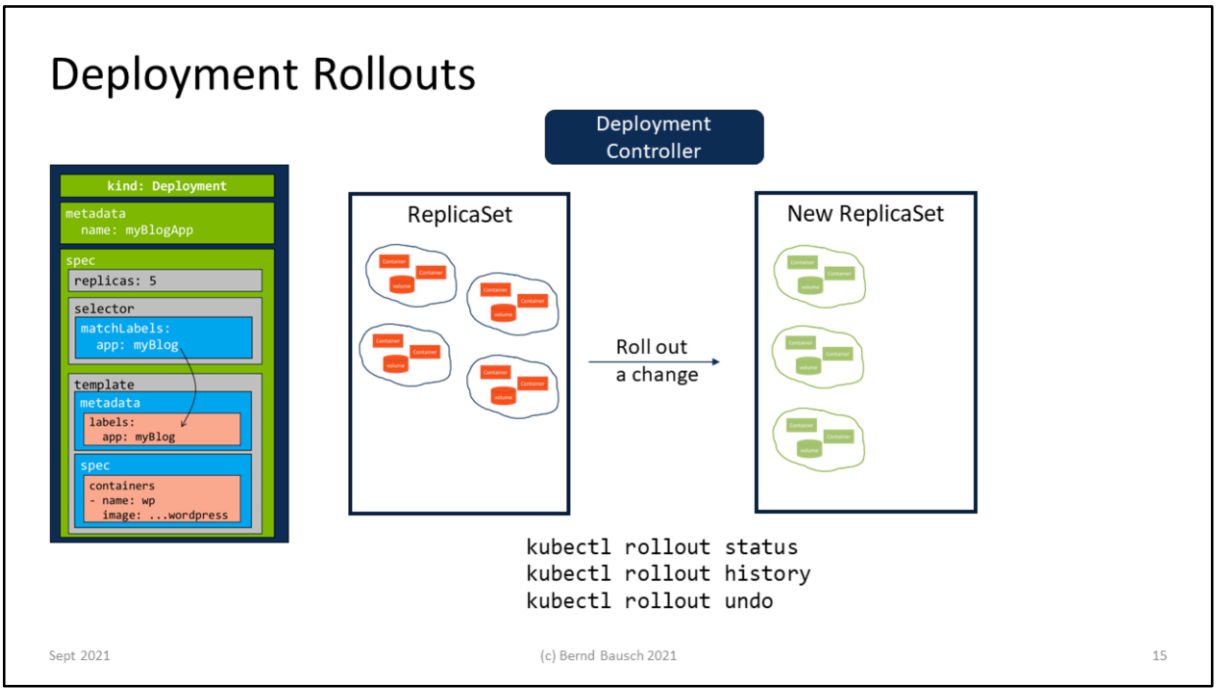
A deployment contains a ReplicaSet and can therefore be scaled in the same way.

Deployments exist because upgrading application software is a common task that must be handled with care. In most cases, it is not possible to just stop the application, install the new version and restart the application. The Deployment controller can perform **rolling updates** of the pods in a ReplicaSet, i.e. it upgrades one pod after the other (or a subset of all pods after the other), so that the application continues delivering its service while it is being upgraded.

Since upgrades can fail or result in an undesirable state of the application, **rollbacks** are another important feature of Deployments.

While an upgrade or rollback is going on, the Deployment controller allows to check on its status and progress. You can also pause and resume an ongoing rollout.

A rollout does not modify the current pods. It replaces them with new pods. In fact, the ReplicaSet in the Deployment is replaced by a new ReplicaSet.



When applying any change to the pod template, the Deployment controller creates a new ReplicaSet with new pods that correspond to the modified template. Using the default deployment strategy *RollingUpdate*, not all pods are replaced at once, but one after the other (or a specified number of pods at once). This means that the application is accessible during the rollout, and that two versions of the application are up and running at the same time. When rollout is complete, none of the red pods remains, but the old ReplicaSet is kept. This allows rolling the change back.

Not all applications tolerate the presence of two versions. In such a case, configure a deployment strategy of *Recreate*. With this strategy, all old pods are terminated before the first new pod is launched. This results in a short service disruption.

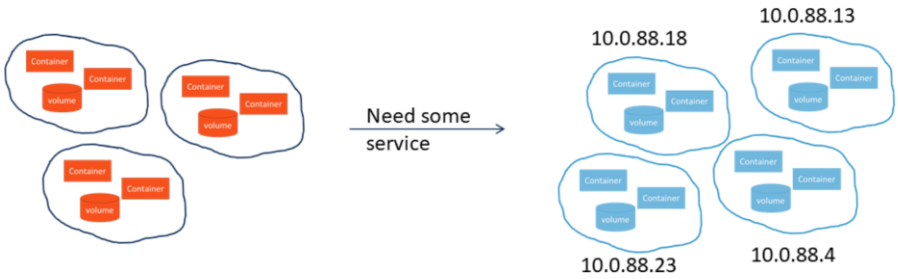
The Rollout can be fine-tuned with parameters in the spec section of the Deployment manifest, for example maxSurge is the maximum total number of pods (the diagram shows seven active pods) or maxUnavailable tells how many pods can be missing at any time (in other words, the minimum total number of pods).

Whenever a rollout occurs, the old ReplicaSet is kept. You can see this by listing all your ReplicaSets with kubectl get rs. The maximum number of old ReplicaSets is also a spec field.

Application Access

Part 1: Services

The Problem

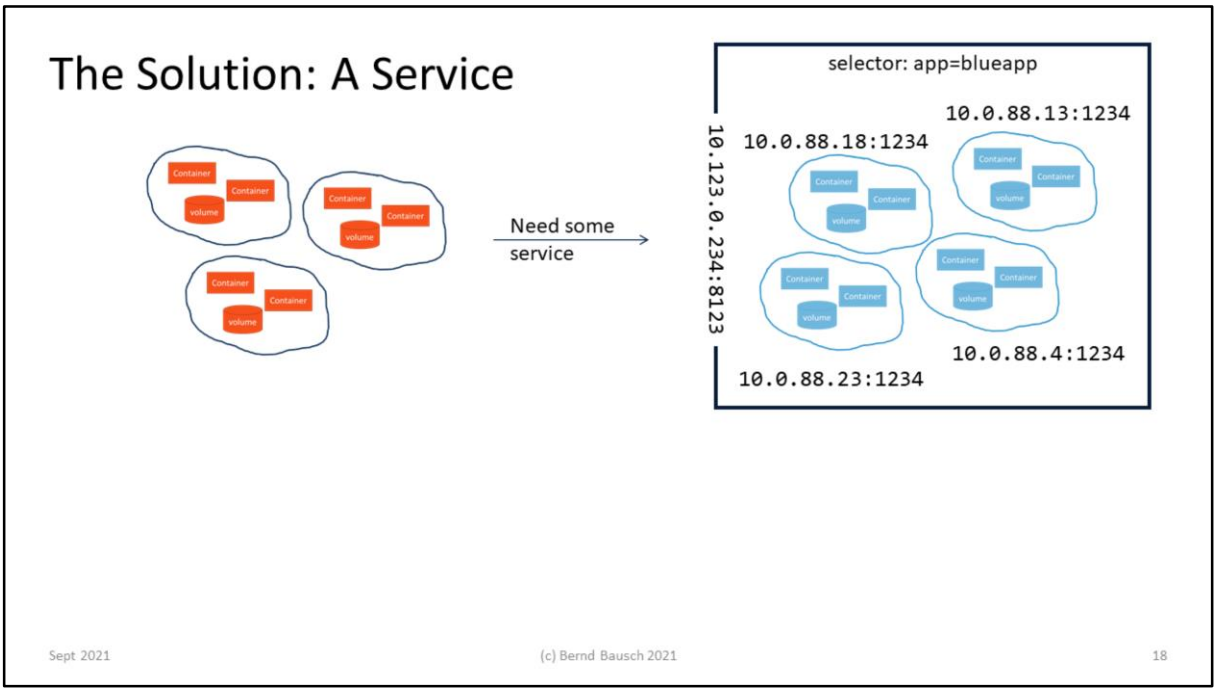


Sept 2021 (c) Bernd Bausch 2021 17

The group of blue pods delivers some service, which the red pods would like to consume. The red pods are not interested in the internal structure of the blue group. For example, it doesn't care whether the blue group scales up or down, and which IP addresses are assigned to the blue pods.

All the red pods want is a service that is collectively provided by the blue application.

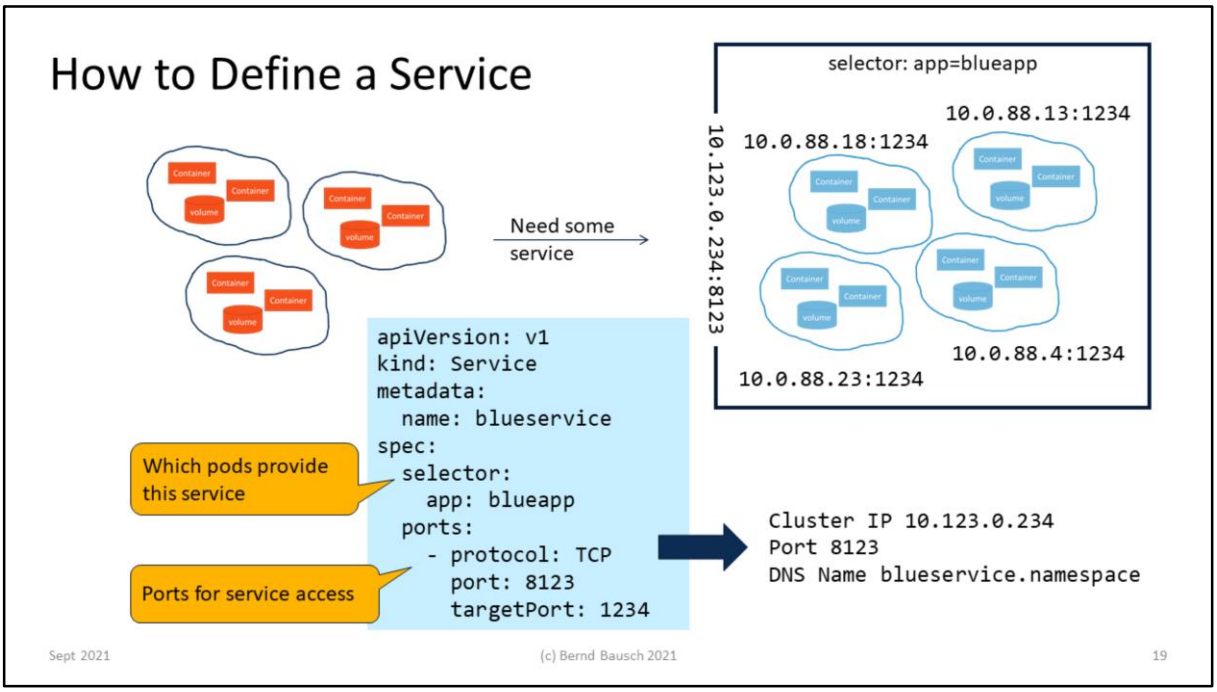
Important: The blue pods all have the same specification - they are identical copies of each other.



The Kubernetes **service** concept makes it possible to address a group of pods as a unit. Essentially, a service consists of two parts: A method of defining the group of pods that provide the service (which pods belong to the blue application), and an address (IP address and port number) by which it can be reached.

In the example, pods provide their service on port 1234. The service is available at a different port, 8123, and makes a single IP address 10.123.0.234 available. Kubernetes forwards accesses to 10.123.0.234:8123 to the blue pods in a round-robin fashion.

The blue pod's IP addresses and ports are named **endpoints**. Blue ports are called **target ports**.



The manifest in the center defines a port. The selector defines the group of blue pods that ultimately provide the service. the ports key in the manifest lists the service and corresponding target ports. In the example, a single service port 8123 is defined; it corresponds to pods's port 1234.

When this manifest is applied, Kubernetes generates an IP address for this port, the so-called Cluster IP. It also creates a DNS name for this address, which consists of the service name and the namespace that it is in. The red pods can use the Cluster IP or the DNS name to access the service.

Service Types

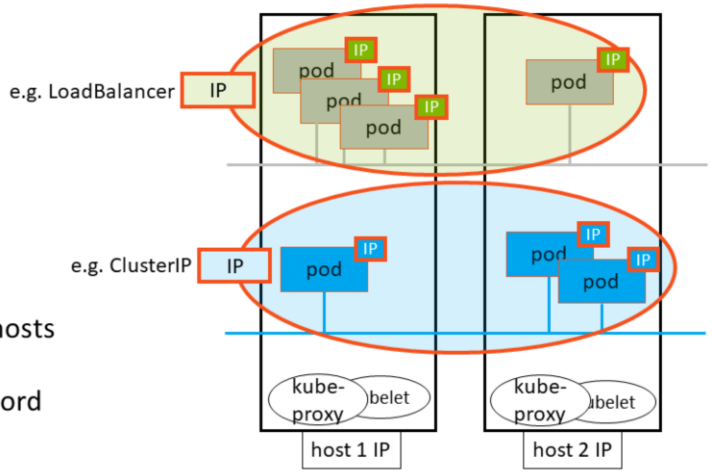
- **Cluster IP**
  Can be used inside cluster
- **NodePort**
  Service IP address is worker host (node) IP address
- **LoadBalancer**
  Sets up cloud provider's Load Balancer
- **ExternalName**
  Creates DNS CNAME record

- **Ingress**
  Not a service type
  Fronts a service and routes traffic over same external IP

How to access an app: Services

Service Types

- **ClusterIP**
  access from other pods
- **NodePort**
  access from outside
  via host's IP address
- **LoadBalancer**
  K8s manages LB across hosts
- **ExternalName**
  Creates DNS CNAME record



Sept 2021 (c) Bernd Bausch 2021 21

Several service types exist, which build on top of each other. The most basic service type is **ClusterIP**; it provides an IP address that can be used inside the cluster, but is not accessible from outside.

A **NodePort** service adds external routing: An external application can access the service through any worker node's IP address and the service port.

You could implement an application with a NodePort service and a load balancer that distributes traffic to the nodes. You would be responsible for the configuration and management of the load balancer. Alternatively, you can configure a **LoadBalancer** service, which creates and manages the load balancer for you. This requires adding load balancer functionality to your Kubernetes cluster; a feature that cloud providers like AWS, Azure or GCP offer out of the box.

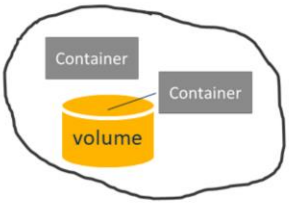
ExternalName sim

Storage

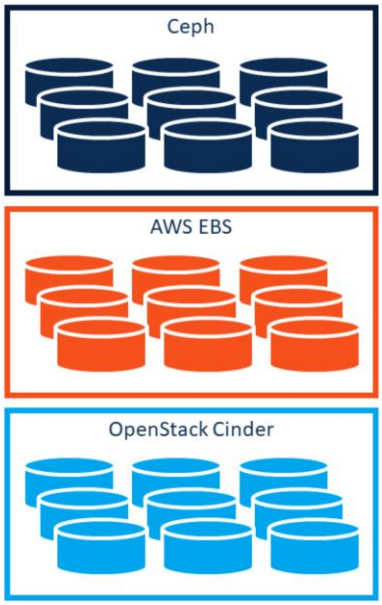
Ephemeral Volumes, Persistent Volumes, Persistent Volume Claims

Kubernetes Volumes

```
apiVersion: v1
kind: Pod
metadata:
  name: website
  labels:
    app: mySite
spec:
  containers:
  - ....
    volumeMounts:
    - mountPath: /srv
      name: myvol
  volumes:
  - name: myvol
    cinder:
      volumeID: xxxyyyzzz
      fsType: ext4
```



- ephemeral
- persistent

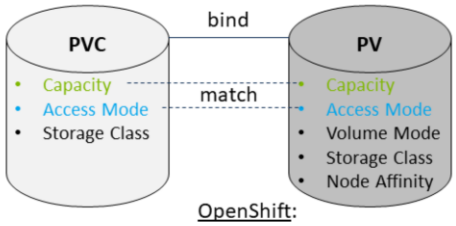


Sept 2021 (c) Bernd Bausch 2021 23

Volumes are independent of containers.

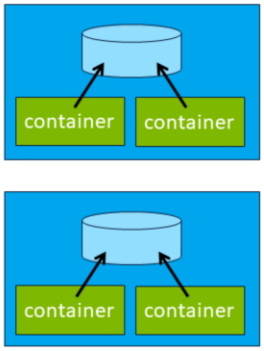
They are declared in the pod manifest under the **volumes** spec. A container in the same pod can then mount this volume (it may be mounted by several containers), i.e. connect it to a directory.
Volumes may be deleted when the pod is deleted. Such volumes are named **ephemeral**. Example: Volume type **emptyDir**, which resides on the same host as the pod and can be compared to the Docker volumes we have encountered. An emptyDir volume can even reside in RAM.
There is a large number of persistent volume types, starting with storage types offered by cloud providers such as AWS EBS, Azure Files, or GCE Persistent Disk. Drivers for many storage solutions in private datacenters exist, such as Ceph, Cinder, Fibre-Channel, GlusterFS, NFS, iSCSI and many more.
The example pod illustrates a problem. The volume is a specific Cinder volume that must exist before the pod can be created. **Who creates and manages this Cinder volume?** Normally, pod authors just want some storage space with certain characteristics e.g. persistent, fast, cheap, certain size etc, and don't want the additional work and complexity of managing storage space.

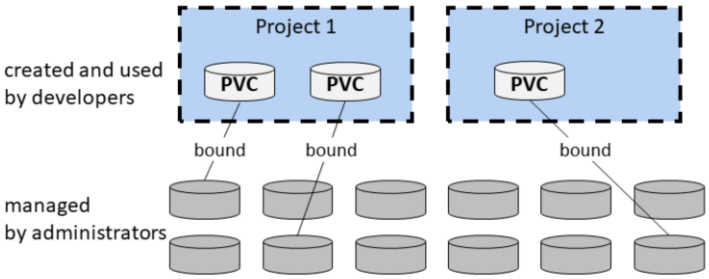
Storage



OpenShift:
The PV bound to the PVC matches two criteria *AccessMode* and *Capacity*

Pods with ephemeral volumes





Example: Two containers sharing a "scratch-pad" volume

```yaml
apiVersion: v1
kind: pod
spec:
    volumes:
      - name: html
        emptyDir: {}
    containers:
      - name: nginx
        image: nginx:alpine
        volumeMounts:
          - name: html
            mountPath: /usr/share/nginx/html
            readOnly: true
      - name: html-updater
        image: alpine
        command: [ "/bin/sh", "-c" ]
        args:
          - while true; do date >> /html/index.html; sleep 10; done
        volumeMounts:
          - name: html
            mountPath: /html
```

Volume declaration

Volume mounted in containers

Sept 2021 (c) Bernd Bausch 2021 25

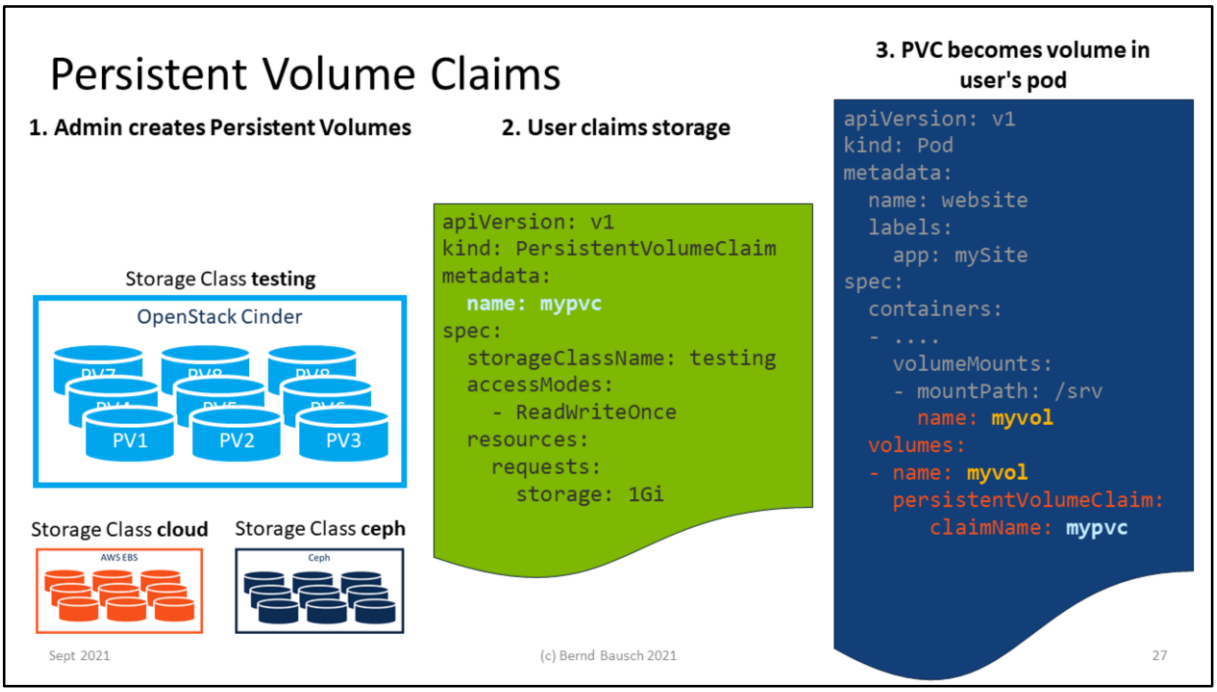
Example: Socket mounted from host

```
apiVersion: v1
kind: pod
spec:
    volumes:
      - name:
        hostPath:
          path: /var/run/docker.sock
          type:
    containers:
      - name: docker
        image: docker
        command: ["sleep"]
        args: ["100000"]
        volumeMounts:
          - name:
            mountPath: /var/run/docker.sock
```

Volume declaration

Volume mounted in container

Sept 2021 (c) Bernd Bausch 2021 26



Pod authors don't want to be bothered with volume management. Rather than creating a specific volume on a specific provider (EBS, Ceph, Cinder, ...), they want to be able to request a volume and leave the details to the administrator or, better, to the system.

Users can be given some control over the location and/or characteristics of their volumes. **StorageClasses** allow administrators to classify available storage according to criteria like speed, provisioning (thick or thin), RAID levels, availability levels, proximity etc. Users can then select the StorageClass that suits their application or budget. In the example, administrators configured a storage class named testing that is backed by an OpenStack Cinder installation, a second storage class for cloud storage, and a third for storage that comes out of a local Ceph cluster.

To request storage without worrying for implementation details, users create a **Persistent Volume Claim**, PVC.

When a user makes a PVC, it is satisfied from a storage pool that administrators made available. This example shows persistent volumes PV1 to PV9 that an administrator created in the *testing* storage class. When the user makes a PVC, Kubernetes looks for a suitable (i.e., large enough) PV. If a PV is found, it is **bound** to the PVC.

The user can then use the claim in a pod as if it were a regular volume.

Dynamic Volume Provisioning

1. Admin creates StorageClass

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: cloud
provisioner: kubernetes.io/aws-ebs
parameters:
  type: io1
  iopsPerGB: "10"
  fsType: ext4
```

2. User claims storage

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mypvc
spec:
  storageClassName: cloud
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

3. PVC becomes volume in user's pod

Sept 2021 (c) Bernd Bausch 2021 28

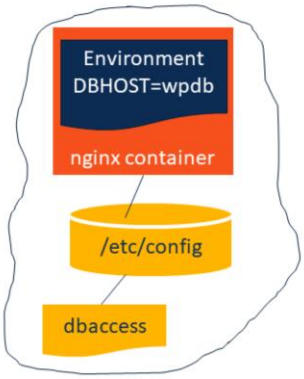
The PVC solution shown on the previous slide has downsides. PVs have to be created in advance. Since the size of PVCs is not known, PVs may be too small, so that PVCs can't be satisfied, or too large, so that storage is wasted (a PVC can not be bound to a partial PV).

To address this problem, many storage backends allow dynamic provisioning.

ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: dbConfig
data:
  dbhost: "wpdb"
  db: |
    dbname="wordpress"
    dbuser="wpuser"
    dbpasswd="pw"
```

```
apiVersion: v1
kind: Pod
metadata:
  name: blog
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    env:
    - name: DBHOST
      valueFrom:
        configMapKeyRef:
          name: dbConfig
          key: dbhost
    volumeMounts:
    - name: wpconfig
      mountPath: /etc/config
      readOnly: true
```




```
volumes:
  - name: wpconfig
    configMap:
      name: dbConfig
      items:
      - key: db
        path: dbaccess
```

Sept 2021 (c) Bernd Bausch 2021 29

A ConfigMap permits configuring a pod from outside. You could do that by creating volumes that correspond to pod's configuration files, or by creating environment variables, but it’s good practice to keep configuration details separate from pod definitions.

The configmap manifest shown here defines a ConfigMap named dbConfig. It defines two configuration keys dbhost and db.

The pod defines how these configuration keys are then used in the application. As an example, the pod shown here copies the value of the dbhost key in an environment variable named DBHOST. The value of the db key is copied into a file in a volume of type configMap, which is mounted at /etc/config. The value of configuration key db is found in the dbaccess file in that volume.

When a value in a config map is modified, the modification is reflected in a configMap-type volume.

Secrets

```
apiVersion: v1
data:
  .dockerconfigjson: <base64-encoded docker config>
kind: Secret
metadata:
  creationTimestamp: "2021-10-02T10:50:50Z"
  name: regcred
  namespace: default
  resourceVersion: "24379"
  uid: 3e42b33e-bdd7-487a-b43d-655d36b835cd
type: kubernetes.io/dockerconfigjson
```

```
apiVersion: v1
kind: Pod
metadata:
  name: blog
  spec:
    containers:
    - name: nginx
      image: nginx:1.14.2
      volumeMounts:
      - name: dockercred
        mountPath: /etc/docker
        readOnly: true
    volumes:
    - name: dockercred
      secret:
        secretName: regcred
```

```
kubectl create secret docker-registry regcred --docker-username=...  --docker-password=...
        --docker-email=... --docker-server=https://index.docker.io/v1/
```

Sept 2021 (c) Bernd Bausch 2021 30

Generally, a secret is sensitive data such as passwords or cryptographic keys. When a pod needs a secret, for example to authenticate with some service, you could enter the secret value in the pod's manifest. This is a poor choice for security reasons.

Instead, you can create and store a secret in Kubernetes and make it available to the pod either as a volume or in an environment variable. Kubernetes Secrets can be considered a special type of ConfigMap with the purpose of storing confidential information.

By default, Secrets are stored in etcd without encryption. For security, the Kubernetes cluster should be configured for encryption and should enable RBAC to control access to secrets. This won't be covered in the Kubernetes module, but both features are standard in OpenShift.

There are several types of Kubernetes Secrets that feature different options, such as opaque for generic secrets, dockerconfigjson for Docker configuration content (including authentication), and ssh-auth.

Secrets can be created by crafting manifests, but it is probably easier to use the command line as shown in the example.

Like ConfigMaps, Secrets can also be imported to environment variables in the pod manifest.

OpenShift Volume Plugins

Public cloud
AWS Elastic Block Store (EBS)
Azure Disk
Azure File
GCE Persistent Disk

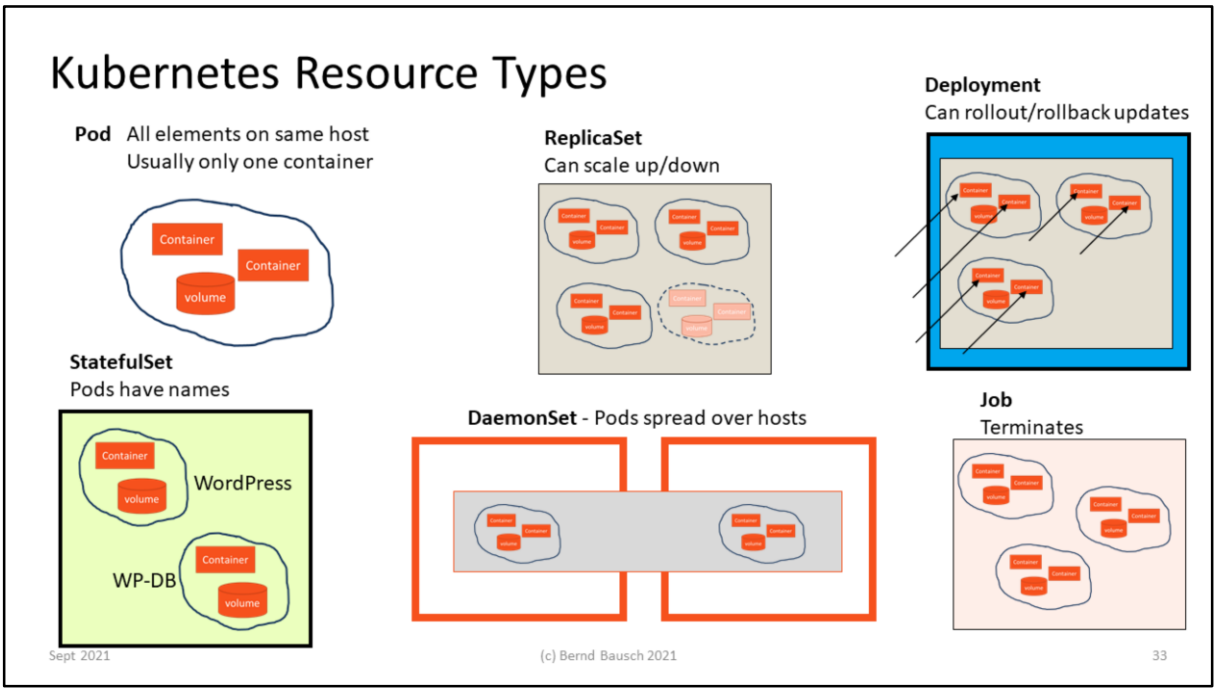
Private Cloud
OpenStack Cinder
OpenStack Manila
VMware vSphere

Private Datacenter
Fibre Channel
iSCSI
NFS

Local
HostPath
Local volume
Red Hat OpenShift Container Storage

Kubernetes Workload Types

Part 2: StatefulSets, DaemonSets, Jobs



This diagram summarizes the abstractions Kubernetes offers for managing applications. They are called *resource types*, or *workload types*. A controller is associated with each of them. They are documented at https://kubernetes.io/docs/concepts/workloads.

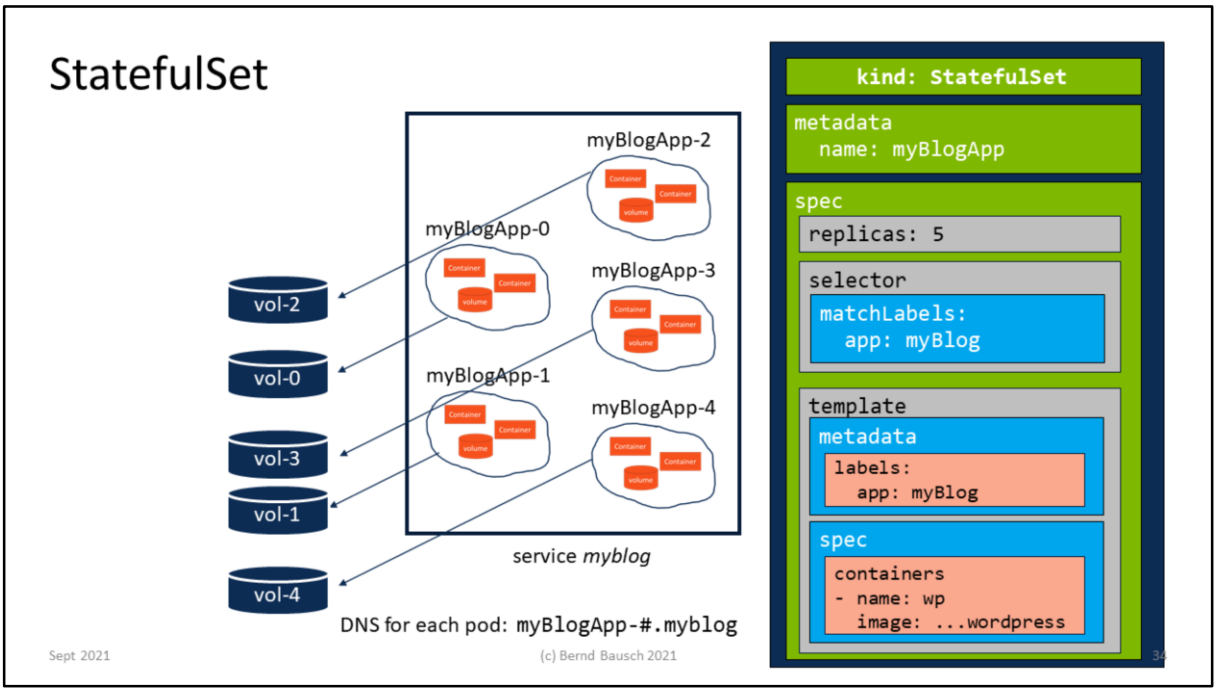
Kubernetes doesn't deal with containers alone. The smallest unit is a **pod**, a collection of containers and volumes that are located on the same container host. In practice, pods often have a single container, though another common pattern is a main container that processes or displays data, and a second container, often called "side-car", which might generate data or perform supporting tasks.

Running single pods in a Kubernetes cluster is possible, but not practiced very often. Instead, most pods are embedded in higher-level resource types, which all have specific purposes.

ReplicaSets are groups of identical pods that can be **scaled up or down** by changing the number of replicas. A typical example is a web site managed by several web server pods. When the load on the web site increases, more web server pods are added (scaling up); when it goes down, some pods are discarded.

Deployments, perhaps the most common way of implementing applications on Kubernetes, each include a single ReplicaSet. A deployment allows the pods in the ReplicaSet to be **upgraded** to more recent image versions. This is a rolling upgrade - not all pods are upgraded at the same time, but only a subset of all pods at a time, in order to avoid disrupting the application. Deployments can report the status of an upgrade, and allow **rolling it back** when it turns out that it introduces problems.

ReplicationControllers, not show on this diagram, are now deprecated precursors of Deployments. They combine the features of ReplicaSets and Deployments in one resource type.



A StatefulSet is a Deployment with additional characteristics. In short, when a pod needs to be replaced (for example because it crashed or when rolling out an update), the replacement pod gets the same state as the old pod. This includes:

- A replacement pod gets the same name as the pod it replaces. Pod names consist of the StatefulSet's name and a number.
- It is connected to the same persistent volume(s) as the pod it replaces
- The number in pod names determines the order in which they are started (counting up) and stopped (counting down).
- When a service is defined for the pods in a StatefulSet, pods have persistent DNS names composed of the StatefulSet name and the service name.
  The service must be headless - i.e. it must not have an IP address itself. This is achieved by setting the ClusterIP in the service's spec to None.

StatefulSet and Volume Claim Template

```yaml
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
  spec:
      terminationGracePeriodSeconds: 10
```

```yaml
containers:
      - name: nginx
        image: k8s.gcr.io/nginx-slim:0.8
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: www
          mountPath: /usr/share/nginx/html
  volumeClaimTemplates:
  - metadata:
      name: www
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "my-storage-class"
      resources:
        requests:
          storage: 1Gi
```

Sept 2021 (c) Bernd Bausch 2021 35

This is a simple StatefulSet copied almost unchanged from https://kubernetes.io/docs/concepts/workloads/controllers/statefulset. The set consists of three NGINX pods that mount a volume named www. In fact, this volume is a volume claim and comes from the highlighted volume claim template.

When this manifest is applied, Kubernetes creates three pods named web-0, web-1 and web-2 after the StatefulSet's name. Each pod makes its own persistent volume claim named www-web-0 etc., after the name of the claim template and of the StatefulSet.

Should one of the pods crash or be deleted, it will be replaced by a pod of the same name and connected to the same PV.

The service name in this manifest references a ClusterIP service without IP address (not shown here), in other words a **headless service**. This service operates a DNS domain default.svc.cluster.local, where *default* is the namespace. It gives each pod has its own persistent DNS domain. Since the service is named nginx, the DNS domains are web-0.nginx etc.

DaemonSet

- Ensure that each worker and/or control node has one pod
- Or a subset of all nodes
- Adding nodes → pods are added, too
- Deleting nodes → pods are deleted, too

- Mostly used for infrastructure applications

Job and CronJob

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi",
"-wle", "print bpi(2000)"]
      restartPolicy: Never
  backoffLimit: 4
```

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
```

Sept 2021 (c) Bernd Bausch 2021 37

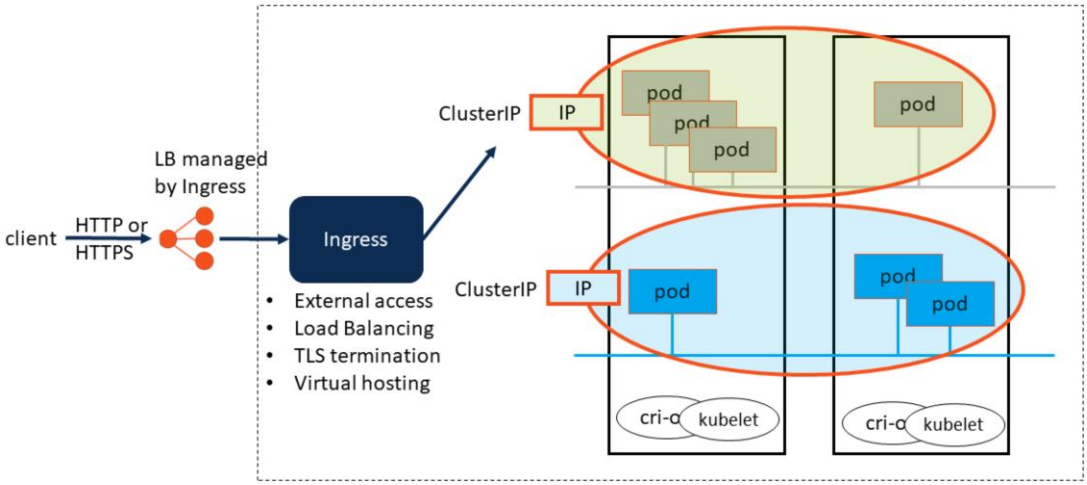
The example on the left comes from https://kubernetes.io/docs/concepts/workloads/controllers/job/. It uses Perl to print 2000 decimal digits of the number π. To see the output, you need the view the pod's log.

On the right, the beginning of a CronJob. The schedule key determines when the job is run. The job itself is declared by the jobTemplate.

Application Access

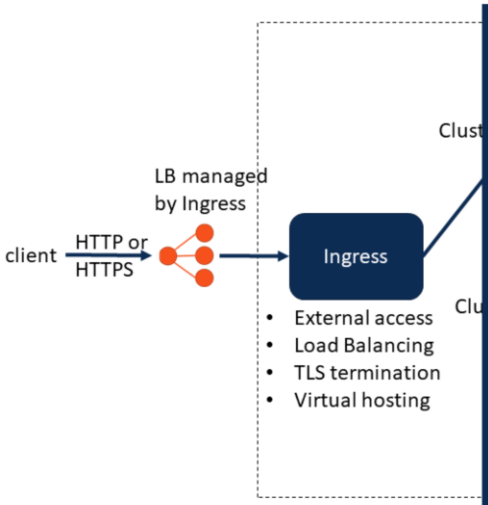
Part 2: Ingress

How to access an app: Service and Ingress



Sept 2021 (c) Bernd Bausch 2021 39

How to access an app: Service and Ingress




```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: mysite.com
    http:
      paths:
      - path: /myblog
        pathType: Prefix
        backend:
          service:
            name: wp-myblog
            port:
              number: 80
```

Option specific to the Ingress controller

Match

Destination

Sept 2021 (c) Bernd Bausch 2021 40

The Ingress manifest compares the Host header to *mysite.com* and the URL path to */myblog*. When a request matches, it is routed to service *wp-myblog*, port 80.

For example, http://mysite.com/myblog/index.html matches.

Helm Charts

What is Helm?

- Templated manifests "**charts**":
  Ensure consistent content
- Package manager:
  Download, install Helm Charts
- Application manager:
  Launch and delete application

Templates

```
kind: Deployment
metadata:
  name: wordpress-mysql
...
 spec:
      containers:
...
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: mysql-pv-claim
```

```
kind: PersistentVolumeClaim
metadata:
  name: mysql-pv-claim
  labels:
    app: wordpress
spec:
...
```

```
kind: Deployment
metadata:
  name: wordpress-mysql
...
 spec:
      containers:
...
      volumes:
      - name: mysql-persistent-storage
        persistentVolumeClaim:
          claimName: {{ .Values.db.pvc.name
}}
```

```
kind: PersistentVolumeClaim
metadata:
  name: {{ .Values.db.pvc.name }}
labels:
    app: wordpress
spec:
...
```

```
db:
  pvc:
    name: mysql-pv-claim
```

values.yaml

Sept 2021 (c) Bernd Bausch 2021 43

On the left, the original WordPress database deployment manifest with hardcoded PVC name, and the corresponding PVC manifest. On the right, both manifests replace the hardcoded name with a variable. The value of this variable can be set on the command line when running helm install, or from a values.yaml file.

Hub and Repos

- Helm Charts published in repos
- Catalog of charts `http://artifacthub.io`

```
helm repo add bitnami https://charts.bitnami.com/bitnami
helm repo list
helm install myblog bitnami/wordpress
helm delete myblog
```