Kubernetes

Containers orchestration

Hélène Coullon, Eloi Perdereau

IMT Atlantique

Table of contents

Reference book

This course is highly inspired by the book **“Kubernetes in action”** written by Marko Lukša

Available online *http://sutlib2.sut.ac.th/sut_contents/H173702.pdf*

Setup your Kubernetes cluster with GKE

Follow this tutorial

Why Kubernetes?

Problems of micro-services architectures

with bigger numbers of micro-services and increasingly complex data centers to deploy them

- difficult to correctly configure and deploy the overall system
- difficult to manage the lifecycle of microservices
- difficult to keep the overall system running

Need for automation and orchestration

- automatic scheduling of micro-services on servers
- automatic configuration and deployment (Docker already helps on this)
- automatic supervision and fault-tolerance

Kubernetes

"Google is one of only a few companies in the world that runs hundreds of thousands of servers and has had to deal with managing deployments on such a massive scale"

- Borg and Omega has been designed for a decade internally
- 2014 Google introduced Kubernetes, an open-source system

Principles

- based on Linux containers
- deploy and manage containerized applications
- no need to know the internal code of the application

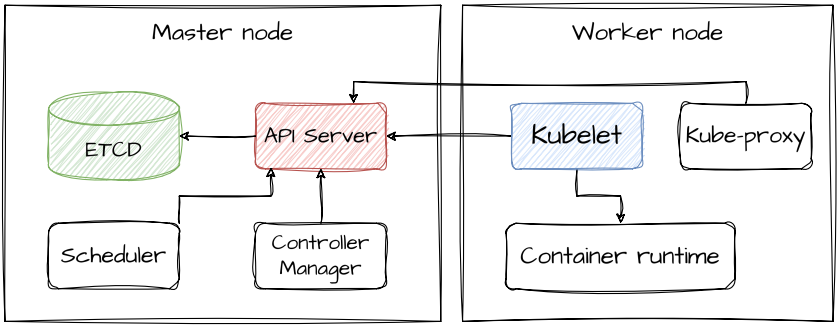
Other containers orchestrators: Nomad (Hashicorp), Docker Swarm

Understanding a bit of internals

Kubernetes architecture

As already mentioned a Kubernetes cluster is composed of at least one master node, and worker nodes.

Let's see how each node works in more details



Master node

Master node

- Kubernetes API server: a user communicate with this API by using *`kubectl`*. Internal services can also communicate with this API.
- Scheduler: schedules applications on worker nodes.
- Controller manager: control all resources of a Kubernetes cluster (including built-in resources such as pods)
- etcd: is a reliable distributed data store used by Kubernetes to store states of resources and configuration of the cluster (more details)

Worker node

- Container runtime: Docker, rkt or another
- Kubelet: talks to the API server and manages containers on the node
- kube-proxy: load-balances network traffic between application components

An introduction of concepts through Pods

What is a pod and why do we need it?

basic building block of Kubernetes that co-locates groups of containers with partial isolation

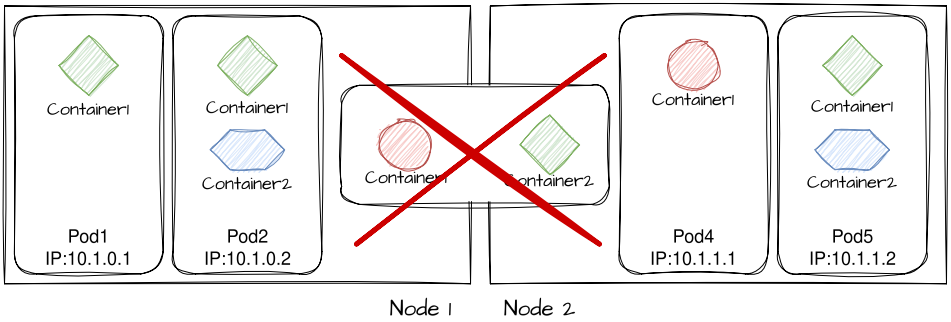
- same Linux namespaces
- same hostname and network interface
- fully isolated filesystem (image dependent)
- shared volumes possible

Idea

- Put tightly related containers (processes) in the same pod
- “Sidecar” containers that should run close to the main container
- examples: data collector, communication adapters etc.

How to know if I need one or two pods?

- do the containers need to be run together or can they run on different hosts?
- do they represent a single whole or are they independent?
- must they be scaled together or individually?

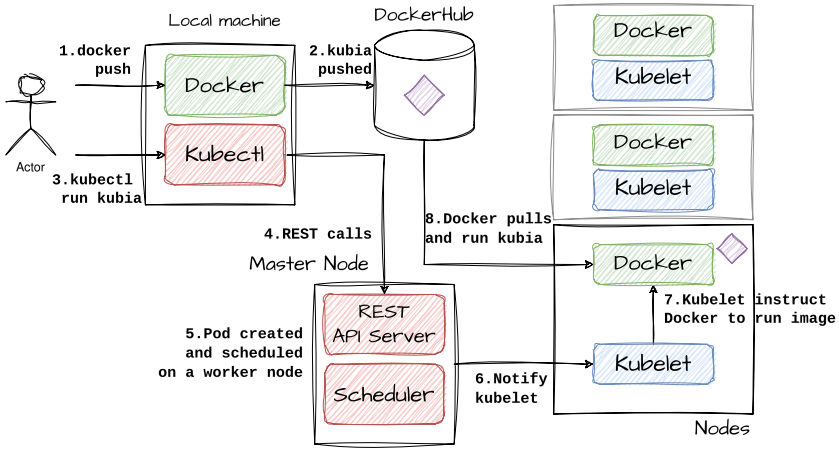


Tutorial

Follow this tutorial

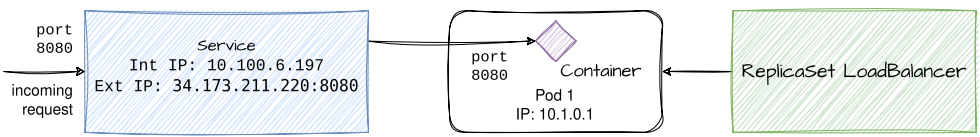
Running a pod

1. We have created a pod through a command line ***kubectl***



Manually exposing a service

2. We have manually exposed the pod through a service of type *loadbalancer*



Resource definition: Manifest

As from Docker to Docker Compose, when handling complex applications we do not want to manually use the `kubectl` command line interface!

YAML is the markup language used by Kubernetes to define required resources. Resources definitions are also known as manifests. This also presents the advantage of being easily “sharable” and “gitable”

A resource is always specified by two blocks of information:

- metadata: name, namespace, labels, and other information about the resource (e.g., pod)
- spec: description of the resource’s content (containers, volumes, etc.)

A third part appears only when the resource is running

- status: current information about the running resource (status, IP, etc.)

Example of a pod definition

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
```

```
> kubectl create -f kubia-manual.yaml
```

Tutorial

Follow this tutorial

Organizing pods with labels

As the number of pods increases we need to categorize them

Let's imagine a micro-service app:

- 20 services
- 5 of them are replicated 5 times
- three versions may be running: stable, beta, canary

We have a total of 120 services running in our Kubernetes cluster!

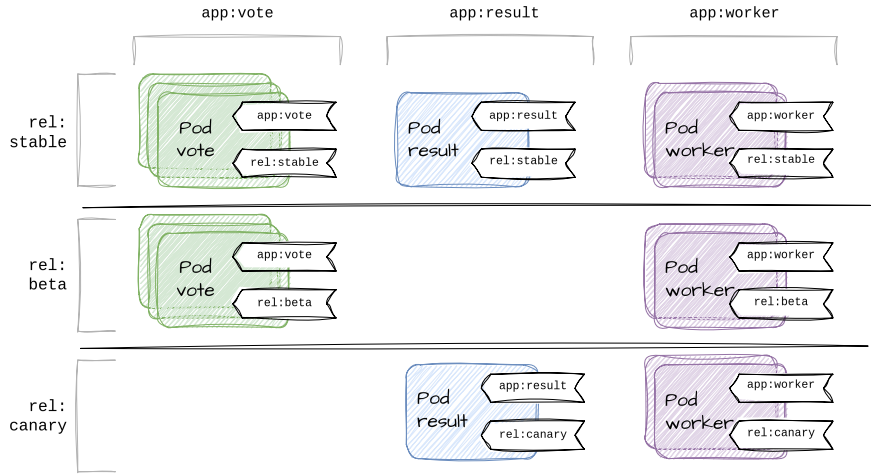
Labels

A label is an arbitrary key-value pair you attach to a resource

Labels selector

resources are filtered based on whether they include the label specified in the selector

Organizing pods with labels



Example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kubia-manual-v2
  labels:
    creation_method: manual
    env: prod
spec:
  containers:
  - image: luksa/kubia
    name: kubia
    ports:
    - containerPort: 8080
      protocol: TCP
```

Label selector

A label selector can select resources based on whether the resource

- Contains (or doesn't contain) a label with a certain key
- Contains a label with a certain key and value
- Contains a label with a certain key with a value not equal to the one you specify

Note that labels can also be used to categorize worker nodes in the cluster and then schedule pods on specific nodes.

Annotations

Annotations are also key-value pairs attached to resources but do not have any selector.

They are used to give much larger information about resources and are usually used by external tools.

Namespaces

Labels offer a way to categorize resources.

What if we more strictly want to separate them? Typically if we want to operate on a subgroup only.

In kubernetes you can group objects (resources) into namespace

- they are not Linux namespaces! (no process isolation by default)
- it is a scope for object names
- same names for resources can be used in different namespaces

Existing namespaces

- *`kube-system`*: this Namespace is used for system processes like etcd, kube-scheduler, etc. Do not modify or create any objects in this Namespace, as it is not meant for users.
- *`kube-public`*: this namespace houses publicly accessible data (e.g., a ConfigMap which stores cluster information like the cluster's public certificate for communicating with the Kubernetes API).
- *`default`*: every namespaced Kubernetes object that is created without specifying a namespace goes to the Namespace defined in your client's configuration. If none are set, objects go to the default Namespace.
- others

Tutorial

Follow this tutorial

Replicas and controllers

Liveness probes

Kubelet on worker nodes are responsible for keeping containers running if they crash. But sometimes the app stops working without a process crash!

Liveness probes

Kubelets can also check if a container is still alive. A liveness probe can be specified for each container in a pod specification.

Four mechanisms are available to define a liveness probe (Details here)

- *`HTTP GET`* probe performs an HTTP GET request on the container's IP address. According to the response code the container is known healthy or not.
- *`GRPC call`* probe tries to call a remote function in the container.
- *`TCP socket`* probe tries to open a TCP connection to the specified port of the container. The connection is established or not.
- *`Exec`* probe executes a command inside the container and checks the command's exit status code.

Note: liveness probes are different from readiness probes

Example

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kubia-liveness
spec:
  containers:
  - image: luksa/kubia-unhealthy
    name: kubia
    livenessProbe:
      httpGet:
        path: /
        port: 8080
```

Tutorial

Follow this tutorial

Do not handle pods manually!

We have seen that Kubelets can restart failing containers through liveness probes.

No more kubelet?

What if a worker node crashes? No more kubelet are available to handle containers...

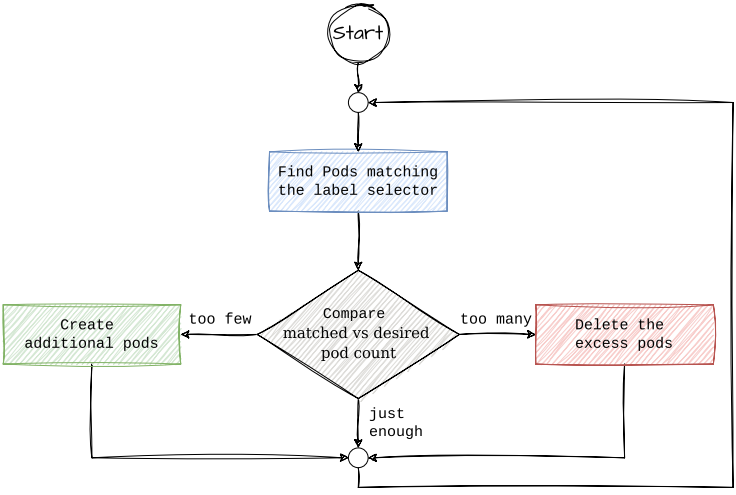
For this reason, in real-life pods are not manually created!

Higher-level resources

higher-level resources are used to create and manage pods such as *`ReplicaSet`*

Higher-level resources are associated with controllers running on the master node responsible for maintaining the state of resources and applications. If a worker node crashes, associated resources will be restarted on another node.

Controller's reconciliation loop



ReplicaSets

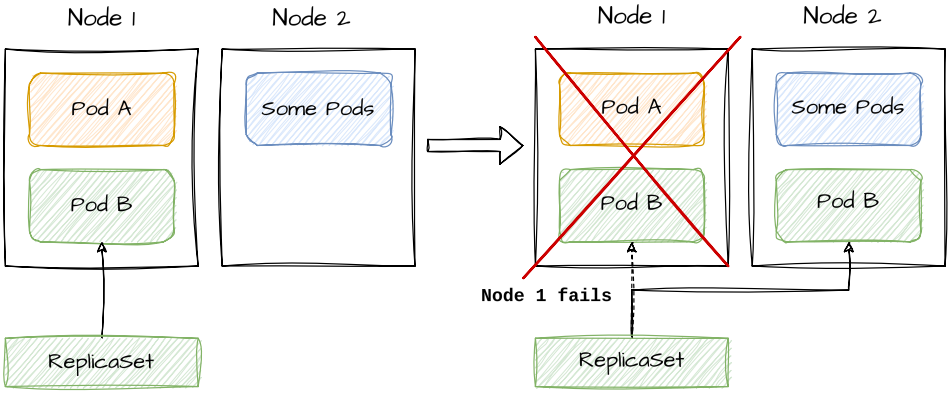
A Replicaset has three essential parts:

- A label selector, which determines what pods are in the replicaset's scope
- A replica count, which specifies the desired number of pods that should be running
- A pod template, which is used when creating new pod replicas

Important notes:

- Changes to the pod template do not affect existing pods
- Changing the label selector makes the existing pods fall out of the scope of the replicaset, so the controller stops caring about them

Replicaset


Node 1
Node 2
Pod A
Some Pods
Pod B
ReplicaSet
Node 1
Node 2
Pod A
Some Pods
Pod B
Pod B
Node 1 fails
ReplicaSet

Example

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    matchLabels:
      app: kubia
  template:
    metadata:
      labels:
        app: kubia
    spec:
      containers:
      - name: kubia
        image: luksa/kubia
```

Label selector expressiveness

The label selector of a replicaset is composed of

- a *key*
- an *operator*
- the *values* to match

An operator can be

- In: value must match one of the specified values
- NotIn: value must not match any of the specified values
- Exists: the pod must include a label with the specified key (the value isn't important and should not be specified)
- DoesNotExist: the pod must not include a label with the specified key (the value isn't important and should not be specified)

Example

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: kubia
spec:
  replicas: 3
  selector:
    matchExpressions:
      - key: app
        operator: In
        values:
         - kubia
  template:
    ...
```

Tutorial

Follow this tutorial

Services and ingresses

Why services?

What is needed:

- Pods need ways to find other pods
- External clients need ways to contact applications hosted on pods

Problem:

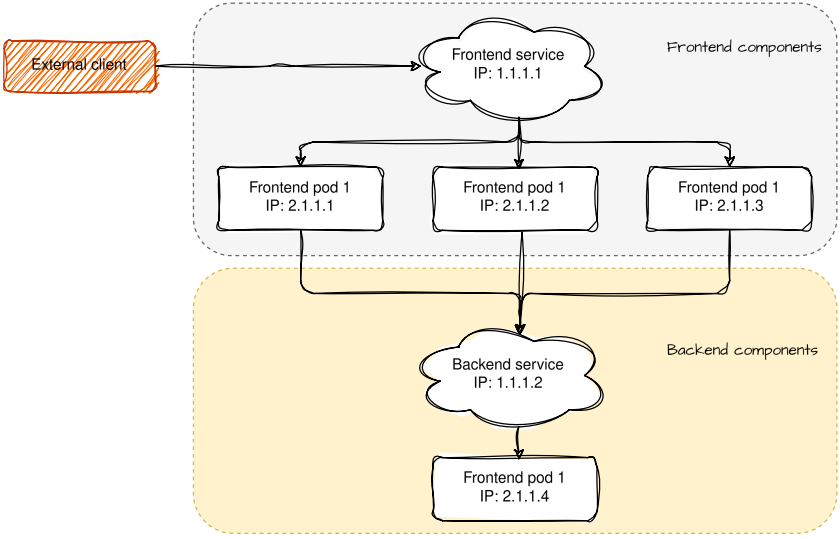
- pods are ephemeral
- the IP address of a pod is known after being scheduled to a node
- horizontal scaling induces that multiple pods may provide the same app

Service

A Kubernetes Service is a resource you create to make a single and constant point of entry to a group of pods providing the same service.

- each service has an IP address and port that never change while the service exists
- clients can open connections to that IP and port, connections are then routed to one of the pods

Example

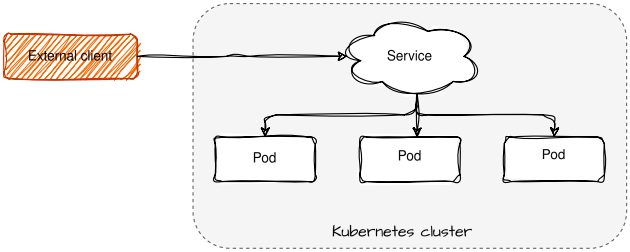

External client
Frontend service
IP: 1.1.1.1
Frontend components
Frontend pod 1
IP: 2.1.1.1
Frontend pod 1
IP: 2.1.1.2
Frontend pod 1
IP: 2.1.1.3
Backend components
Backend service
IP: 1.1.1.2
Frontend pod 1
IP: 2.1.1.4

Example of code

```yaml
apiVersion: v1
kind: Service
metadata:
  name: kubia
spec:
  ports:
  - port: 80
    targetPort: 8080
  selector:
    app: kubia
```

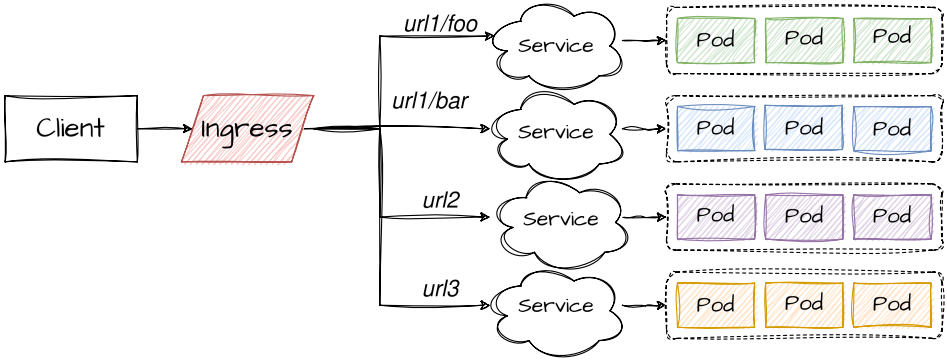
- *port* is the port on which the service will be available
- *targetPort* is the port of the container, the service will forward to this port
- *selector* is the label selector to indicate which pods are concerned by this service

Make a service accessible externally



- type NodePort: each node of the cluster opens a port and redirects the traffic on that port to the underlying service
- type LoadBalancer: makes the service accessible through a load balancer, the load balancer redirects the traffic across all nodes
- Ingress resource: exposing multiple services through a single IP address

Why Ingress resource?



Example of code

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: kubia-ingress
spec:
  rules:
  - host: kubia.example.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: kubia-loadbalancer
            port:
              number: 80
```

Makes sure all HTTP requests received by the Ingress controller at ***kubia.example.com*** at the path **/** will be sent to the ***kubia-loadbalancer*** service on port 80.

Tutorial

Follow this tutorial

Volumes

Volumes in Kubernetes

Pods relies on containers, so as container images are immutable new data within containers are lost when restarted.

Kubernetes volumes

Volumes are part of Pod's definition to declare persistent storage.

The volumes declared in the manifest are accessible to all containers in the pod but have to be mounted in the containers.

Many types of volumes

- emptyDir: Empty directory to store data (persistent at the container level, not at the pods level)
- hostPath: mount a directory from the host worker node
- gitRepo: a volume initialized by cloning a git repository
- nfs: a NFS share is mounted
- cgePersistentDisk, awsElasticBlockStore, azureDisk: to mount specific a Cloud provider-specific storage
- other types of network storage: cinder, cephfs etc.
- configMap, secret, downwardAPI: special types of volumes to expose Kubernetes resources and cluster information to the pod
- etc.

Full list

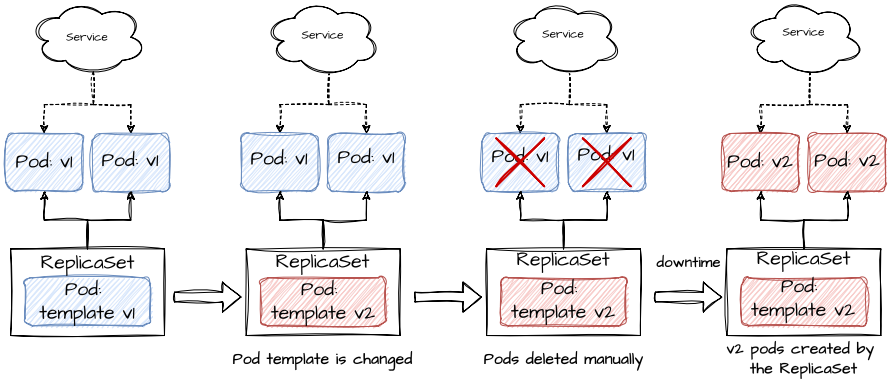
Tutorial

Follow this tutorial

Deployments

Updating pods with downtime

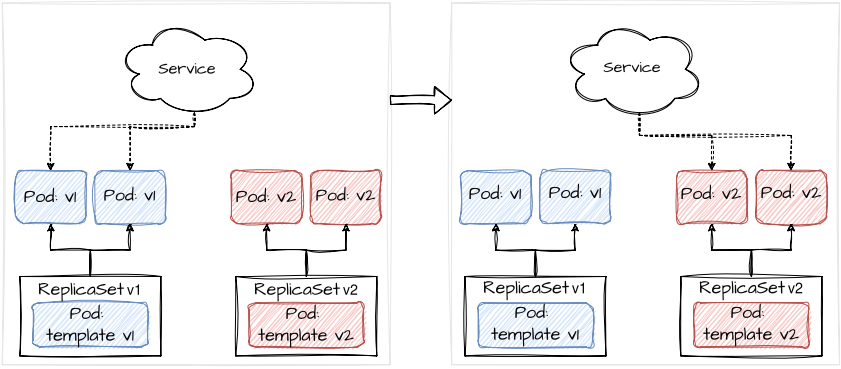
How to move an application from one version to another? The easiest way with downtime



(1) change the template in the replicaset resource, (2) manually delete pods, (3) let the replicaset start new pods with the new version

Updating pods without downtime

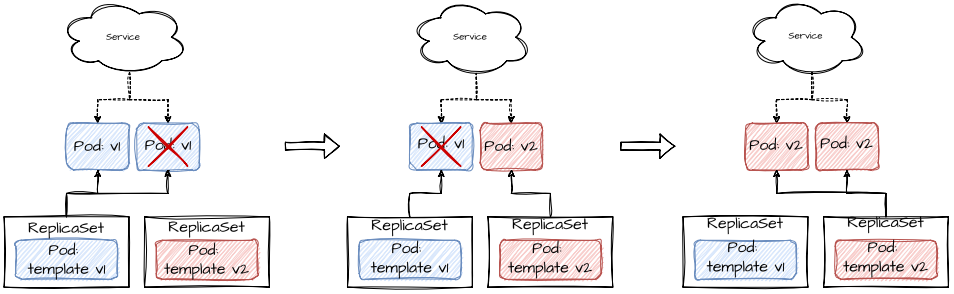
Harder way without downtime, with high hardware cost: blue-green deployment



(1) create a second replicaset with the new version of pods, (2) change the label selector of the service

Rolling update (without downtime)

Even harder way without downtime, less hardware cost



(1) change label selector of the service, (2) scale to 1 the second replicaset and -1 the first replicaset, (3) etc.

Difficult and error prone process!

Deployments

Deployments automate all this for you!

Tutorial

Follow this tutorial

Persistent volumes

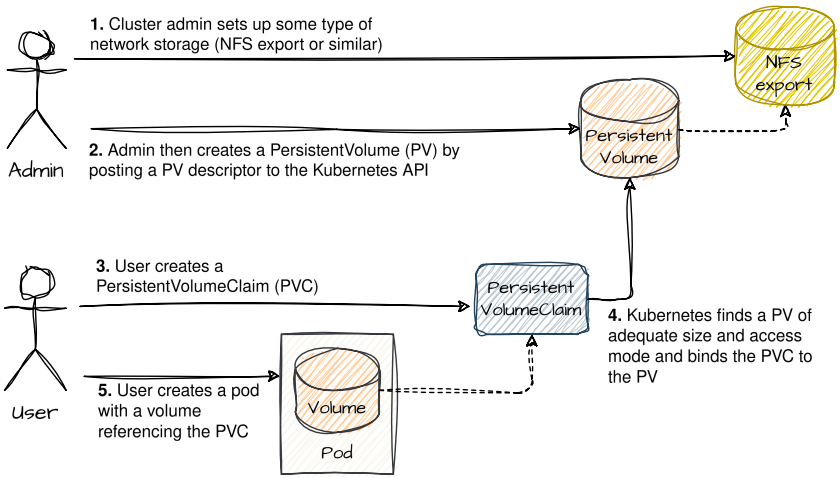
Persistent volume claims

Problem

Volumes explored so far have required the pod developer to know the network storage infrastructure available in the cluster.

This is not the philosophy of Kubernetes. For example, the developer does not have to know the hardware of nodes.

Persistent volume claims



Dynamic allocation of persistent volumes

The problem with the above method is that the administrator needs to provision storage in advance!

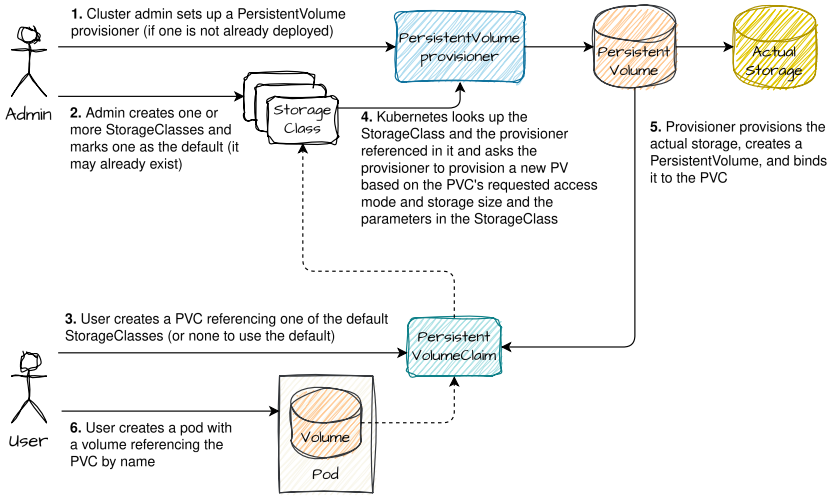
Dynamic allocation of PV - for administrators

- creates a *`PersistentVolume`* provisioner instead of a PV
- defines one or more *`StorageClass`* to let users choose the type of PV they want in their PVC

PV provisioners are already available for popular Cloud providers!

There also are already existing StorageClass objects!

Dynamic PVC


1. Cluster admin sets up a PersistentVolume provisioner (if one is not already deployed)
PersistentVolume provisioner
Persistent Volume
Actual Storage
Admin
2. Admin creates one or more StorageClasses and marks one as the default (it may already exist)
Storage Class
4. Kubernetes looks up the StorageClass and the provisioner referenced in it and asks the provisioner to provision a new PV based on the PVC's requested access mode and storage size and the parameters in the StorageClass
5. Provisioner provisions the actual storage, creates a PersistentVolume, and binds it to the PVC
3. User creates a PVC referencing one of the default StorageClasses (or none to use the default)
Persistent VolumeClaim
User
6. User creates a pod with a volume referencing the PVC by name
Volume
Pod

Tutorial

Follow this tutorial

Configuration Management

Rationale for Configuration Management

Real projects have hundreds or thousands objects, can be more than 100k lines of YAML.

But no mechanism for

- Parametrization
- Reuse, composition of manifests
- Generation with loops

In a word: *no abstraction*.

Helm: Package Manager for Kubernetes

- Template YAML
- Package manifests into ”charts”
- Simple sharing
- Manages releases, upgrades

```
helm install ...
helm upgrade ...
```

Example of Redis chart

Kustomize: Customizing Manifests Without Templating

- Plain YAML
- Customize Kubernetes objects
- Patch/Overlay approach
- Built into *kubectl*

Main usecase: *variants* of manifests for different environments.

```
namespace: dev
resources:
- ../../base
patches:
- path: deployment-dev.yaml
- path: service-dev.yaml
```

Kustomize: Customizing Manifests Without Templating

- Plain YAML
- Customize Kubernetes objects
- Patch/Overlay approach
- Built into *kubectl*

Main usecase: *variants* of manifests for different environments.

```
namespace: dev
resources:
- ../../base
patches:
- path: deployment-dev.yaml
- path: service-dev.yaml
```

```
├── base
│   ├── deployment.yaml
│   ├── service.yaml
│   └── kustomization.yaml
└── overlays
    ├── dev
    │   ├── deployment-dev.yaml
    │   ├── service-dev.yaml
    │   └── kustomization.yaml
    └── prod
        ├── deployment-prod.yaml
        ├── service-prod.yaml
        └── kustomization.yaml
```