Kubernetes Administration from Zero to (junior) Hero

László Budai – Component Soft Ltd.

Agenda

1. Introduction
2. Accessing the kubernetes API
3. Kubernetes workloads
4. Accessing applications
5. Volumes and persistent storage

Introduction

- Cloud computing in general
- Cloud native computing
- Kubernetes overview
- Kubernetes architecture

Cloud computing in general

- a model for enabling ubiquitous network access to a shared pool of configurable computing resources*
  - resources (compute, storage, network, apps) as services
    - resources are allocated on demand
      - scaling and removal also happens rapidly (seconds-minutes)
    - multi-tenancy
      - share resources among thousands of users
      - resource quotas
  - cost effective IT
    - Pay-As-You-Go model
      - pay per hour/gigabyte instead of flat rate
    - maximized effectiveness of the shared resources
      - maybe over-provisioning
    - lower barriers to entry (nice for startups)
      - focus on your business instead of your infrastructure

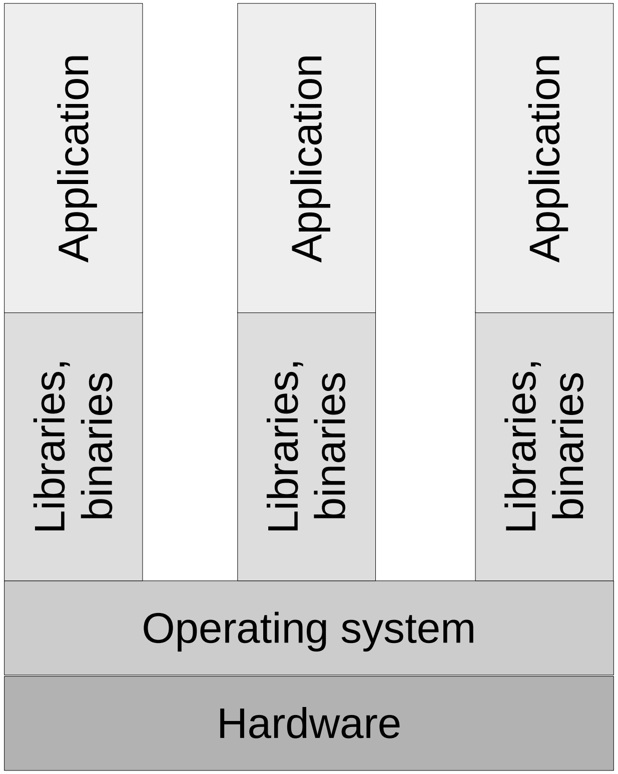
*definition by NIST

Cloud native computing

- a new computing paradigm that is optimized for modern distributed systems environments capable of scaling to tens of thousands of self healing multi-tenant nodes.
- Main properties:
  - Container packaged – containers represents an isolated unit of application deployment.
  - Dynamically managed - actively scheduled and actively managed by a central orchestrating process.
  - Micro-services oriented - loosely coupled with dependencies explicitly described (e.g. through service endpoints).

Application containers

- OS level virtualization – OS partitioning (virtual OS vs virtual HW)
- Allows us to run multiple isolated user-space application instances in parallel.
- Instances will have:
  - Application code
  - Required libraries
  - Runtime
- Self sufficient – no external dependencies
- Portable
- Lightweight
- Immutable images



Container orchestration

- tools that are providing an enterprise-level framework for integrating and managing containers at scale.
- aim to simplify container management
  - a framework for defining initial container deployment
  - availability
  - scaling
  - networking
- Docker Swarm
- Mesosphere Marathon
- Kubernetes

Kubernetes

- Kubernetes – ancient Greek word for helmsman or pilot of the ship
- Initially developed by google
- Has its origins in Borg cluster manager
- “Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.”
- Places containers on nodes
- Recovers from failure
- Basic monitoring, logging, health checking
- Enables containers to find each other

Kubernetes concepts

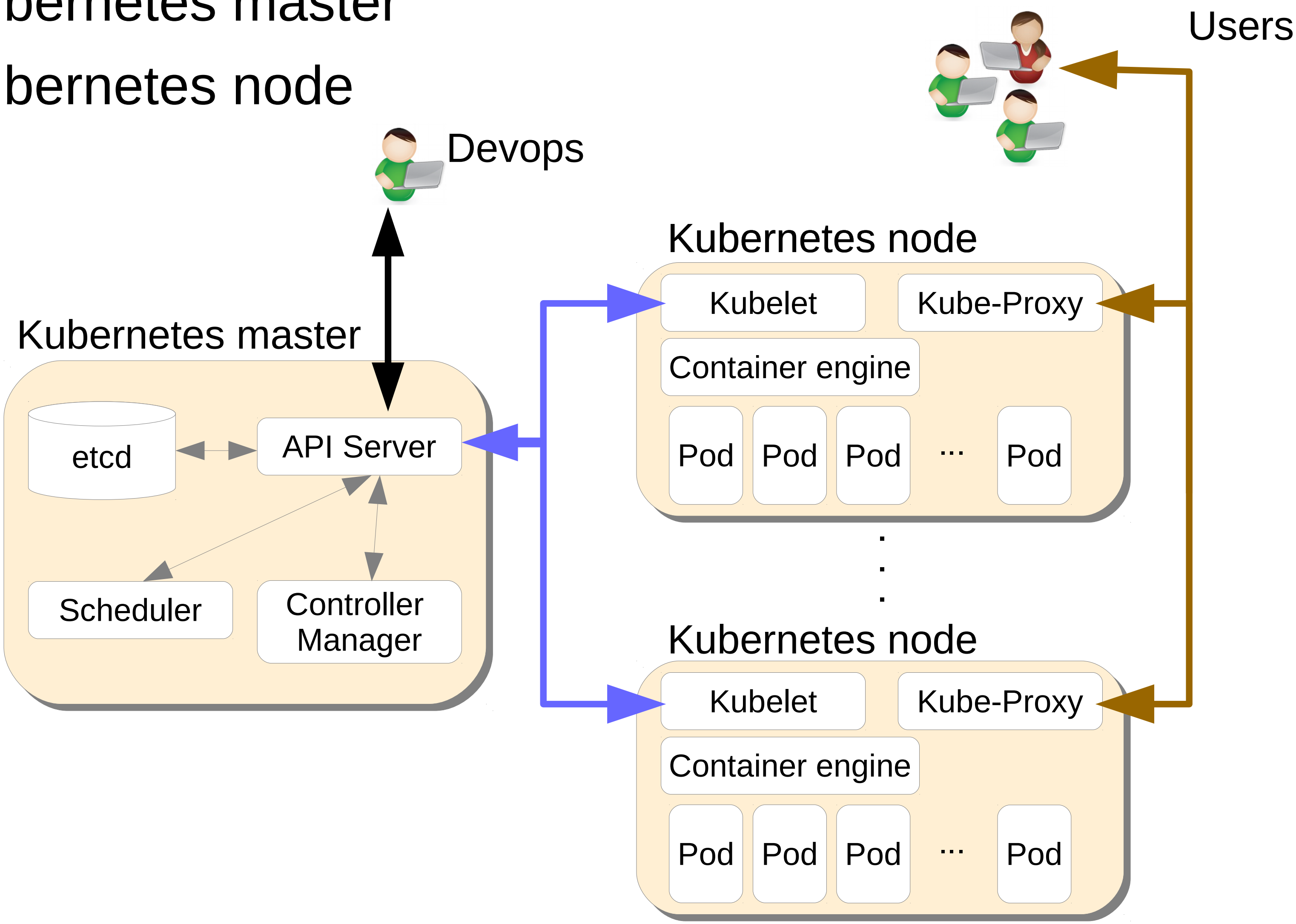
- Kubernetes Master – maintains the desired state for the cluster
- Kubernetes Node – runs the applications
- Kubernetes objects - abstractions that represent the state of the cluster.
  - A “record of intent” - a desired state of the cluster
  - Objects have
    - Spec – describes its desired state
    - State – describes the actual state; updated by Kubernetes.
    - Name – client provided; unique for a kind in a namespace, can be reused
- Namespaces – virtual clusters; provides a scope for names.
- Labels – key-value pairs attached to objects
- Label selector – is the core grouping primitive
- Annotations – attach arbitrary non-identifying metadata to objects

Kubernetes objects categories

- Workloads – used to manage and run the containers (Pod, ReplicationController, deployment)
- Discovery & LB – "stitck" workloads together into an externally accessible, load-balanced Service (Service, Ingress).
- Config & Storage – objects we can use to inject initialization data into applications, and to persist data that is external to the containers (Volume, Secret).
- Metadata – objects used to configure the behavior of other resources within the cluster (LimitRange)
- Cluster – objects responsible for defining the configuration of the cluster itself (Namespace, Binding)

Kubernetes architecture

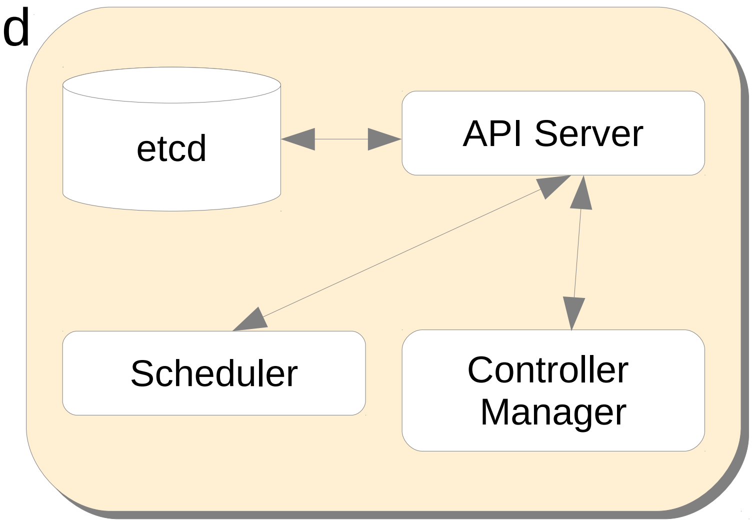
- Kubernetes master
- Kubernetes node



Kubernetes master

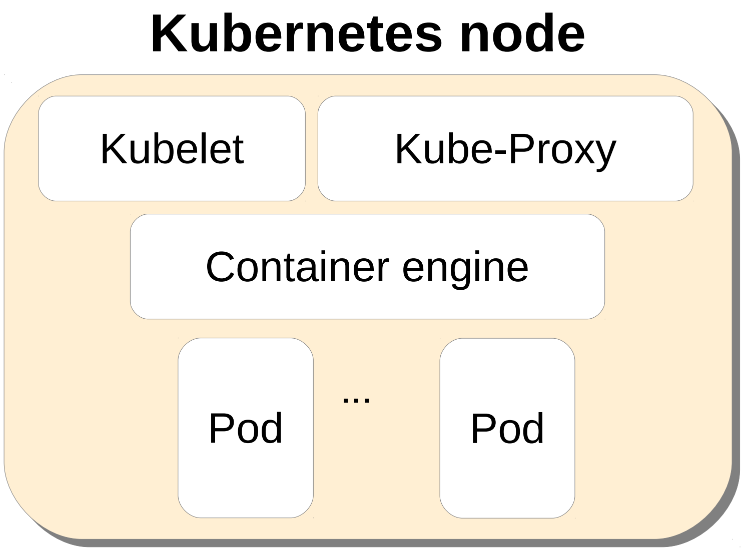
- provide the cluster’s control plane
- kube-apiserver
  - Exposes the Kubernetes API – the front-end for the Kubernetes control plane.
  - Designed to scale horizontally.
- etcd
  - Is the backing store of Kubernetes.
  - Distributed key-value store
- Kube-controller-manager
  - background threads that handle routine tasks
    - Node Controller
    - Replication Controller
    - Endpoints Controller
    - Service Account & Token Controllers
- kube-scheduler
  - Assigns nodes to the newly created pods

Kubernetes master



Kubernetes node

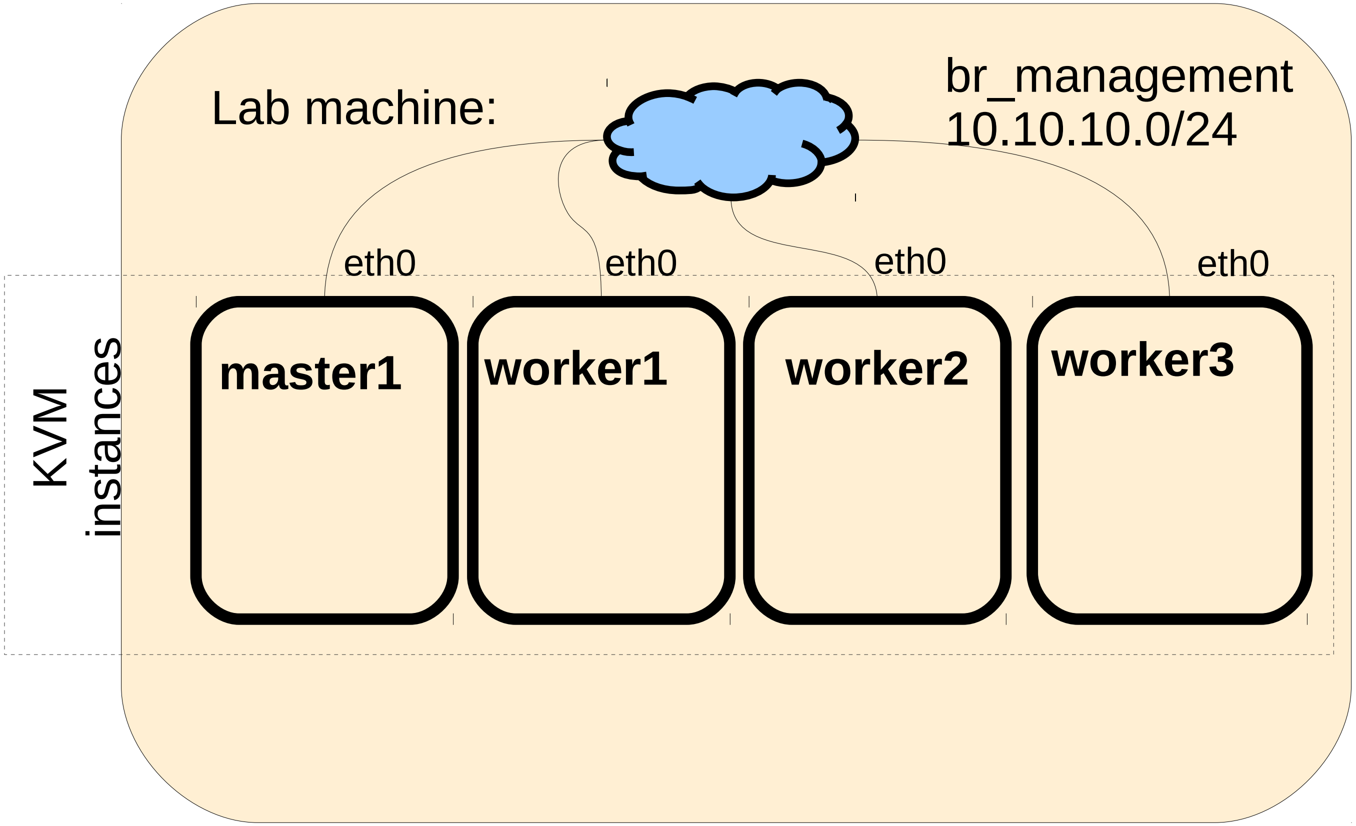
- kubelet - the primary node agent. It watches for pods that have been assigned to its node and:
  - Mounts the pod’s required volumes.
  - Downloads the pod’s secrets.
  - Runs the pod’s containers.
  - Periodically executes any requested container liveness probes.
  - Reports the status of the pod.
  - Reports the status of the node.



- kube-proxy
  - enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding
- Container engine
  - Used to run the containers
  - Docker by default, rkt optionally.
  - Container Runtime Interface – paves the way to alternative runtimes

Exercise 1: The lab environment

- Understanding the classroom environment
- Using **`kubectl`**



2. Accessing the kubernetes API

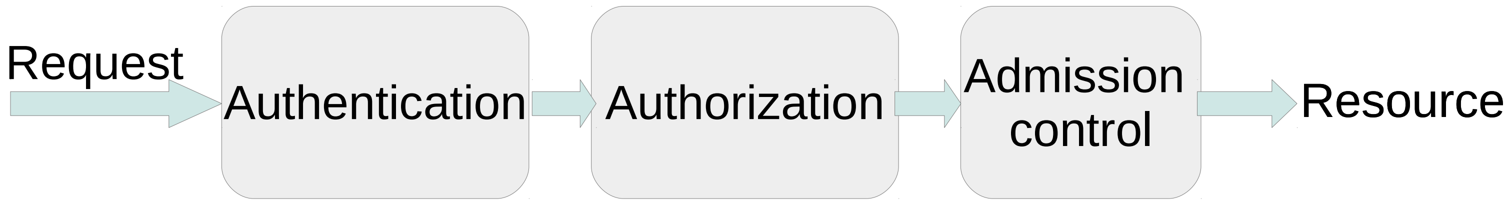
- Ways to access the API
- Controlling access to the API
- Authentication
- Authorization
- Role Based Access Control

Accessing the kubernetes cluster

- kubectl – the command line tool for deploying and managing applications on kubernetes
  - Inspect cluster resources
  - Create, delete, update components
  - Configuration file: ~/.kube/config – information for finding and accessing a cluster
  - bash autocompletion
- Dashboard – web based user interface (add-on)
  - Manage applications
  - Manage the cluster itself
- Direct access to the API
  - HTTP REST

Controlling access to the API

- A request for the API will pass several stages before reaching it



- Authentication – Ensures that the user it is who it pretends to be
- Kubernetes has 2 categories of users:
  - Service accounts – managed by kubernetes
  - Normal users – managed by an independent service
- API requests can be treated as anonymous ones if are not tied to a user or service account.
- Kubernetes uses client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth to authenticate API requests through authentication plugins.

Authorization

- After the user authentication step the request will have to pass the authorization step.
- All parts of an API request must be allowed by some policy → permissions are denied by default.
- Authorization modules
  - Node
  - ABAC – Attribute-based access control
  - RBAC – Role-based access control
  - Webhook

Role Based Access Control

- RBAC allows fine grained rules for accessing the cluster
- allows dynamic configuration of policies through the Kubernetes API.
- uses the “rbac.authorization.k8s.io” API group
- It defines Roles and RoleBindings in order to assign permissions to subjects.
- These permissions can be set
  - Clusterwide – can be used for cluster-scoped resources, non-resource endpoints, namespaced resources across all namespaces
  - Within a namespace.
  - For one single resource.
- Subjects can be users, groups, and service accounts

Roles and ClusterRoles

- RBAC roles contains the rules that represent the permissions
- Permissions are purely additive
- A role can be defined within a namespace, or cluster-wide (ClusterRole)

```
kind: Role
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

- ClusterRoles are not namespaced

Role bindings

- Role binding grants the permissions defined in a role to a subject.
- Permissions can be granted within a namespace with a RoleBinding, or cluster-wide with a ClusterRoleBinding
- A RoleBinding can use a ClusterRole. The rules will apply to the namespace of the binding.

```
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1beta1
metadata:
   name: read-pods
   namespace: development
subjects:
 - kind: User
   name: dave
   apiGroup: rbac.authorization.k8s.io
roleRef:
   kind: ClusterRole
   name: cluster-pod-reader
   apiGroup: rbac.authorization.k8s.io
```

Exercise 2: RBAC

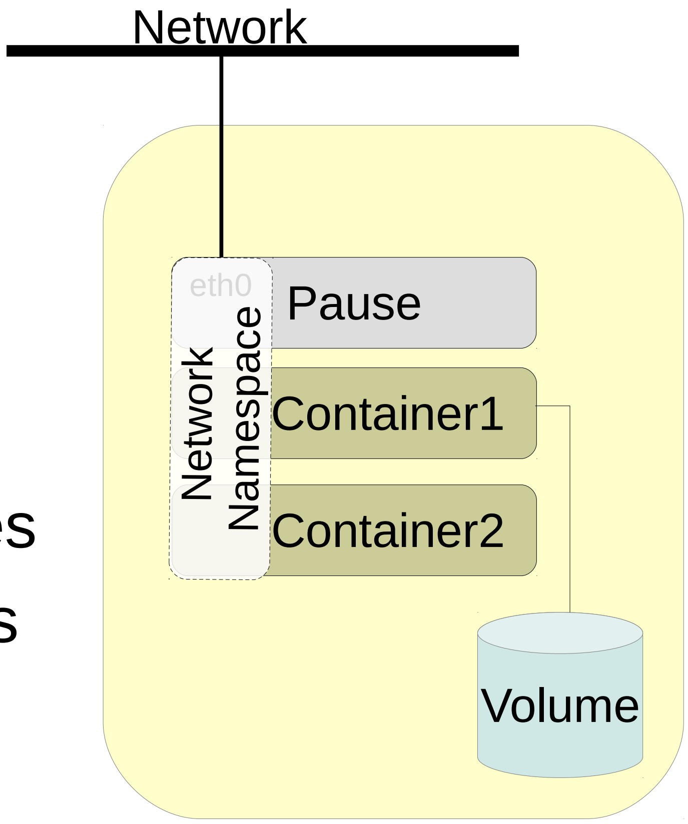
– Use RBAC to control access to the API

3. Kubernetes workloads

- Pod
- Replication controllers
- Deployments, Replica sets
- Jobs and CronJobs
- DaemonSets

The pod

- Pod - the smallest deployable object in the Kubernetes object model.
- It runs a single instance of an application
- Contains
  - One or more application containers
  - Storage resources
  - A unique IP address
  - Options about how the container(s) should run.
- Containers in one pod are sharing the network namespace and storage resources
- A pod is scheduled on a node and remains there until terminated or evicted
- Pods do not self-heal by themselves → controller.



The pod (cont)

- Pod lifecycle:
  - Pending – pod has been accepted by the Kubernetes system, but one or more of the Container images has not been created.
  - Running – has been bound to a node, all of the containers have been created. At least one container is still running (or starting / restarting).
  - Succeeded – all containers have terminated in success, and will not be restarted
  - Failed - All Containers have terminated; at least one has terminated in failure.
  - Unknown – the state of the pod could not be obtained
- Probes – performed by the kubelet on a Container using a handler
  - Probe types – what is testing: readinessProbe, livenessProbe
  - Handler Types – how is testing: ExecAction, TCPSocketAction, HTTPGetAction
  - Probe result: Success, Failure, Unknown
- Restart policy – restarts a pod based on the liveness test result
  - restartPolicy: Always, OnFailure, Never
- Pods are restarted on the same node, only controllers can schedule a new pod on a different node.

Our first Pod

Describe the Pod using a YAML file:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: busybox
spec:
  restartPolicy: OnFailure
  containers:
   - name: busybox
     image: busybox
     command:
       - sleep
     args:
       - "100"
```

Operations on pods

- Create the pod using the kubectl command:
  - kubectl create -f pod1.yaml
- Check the pod status
  - kubectl get pod busybox [-o wide]
  - kubectl get pod --watch
- Get information about the pod
  - kubectl describe pod busybox
  - kubectl get pod busybox -o yaml
- Check the logs of a pod
  - kubectl logs busybox
- Execute a command inside the pod
  - kubectl exec -ti busybox sh
- Delete the pod
  - kubectl delete pod busybox

ReplicaSet

- The ReplicaSet controller simply ensures that the desired number of pods matches its **label selector** exists and are operational
- If the labels of the pod are modified and they do not match the label selector, then a new pod is spawned, the old one stays there.
- The ReplicaSet provide a declarative definition of what a Pod should be and how many of it should be running at a time.

rs1.yaml

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

Working with ReplicaSet

- Create the ReplicaSet
  - kubectl create -f rs1.yaml
- Check the status
  - kubectl get rs [--watch]
  - kubectl describe rs nginx
- Change the number of replicas
  - kubectl scale rs nginx --replicas=3
- Delete the ReplicaSet
  - kubectl delete rs nginx

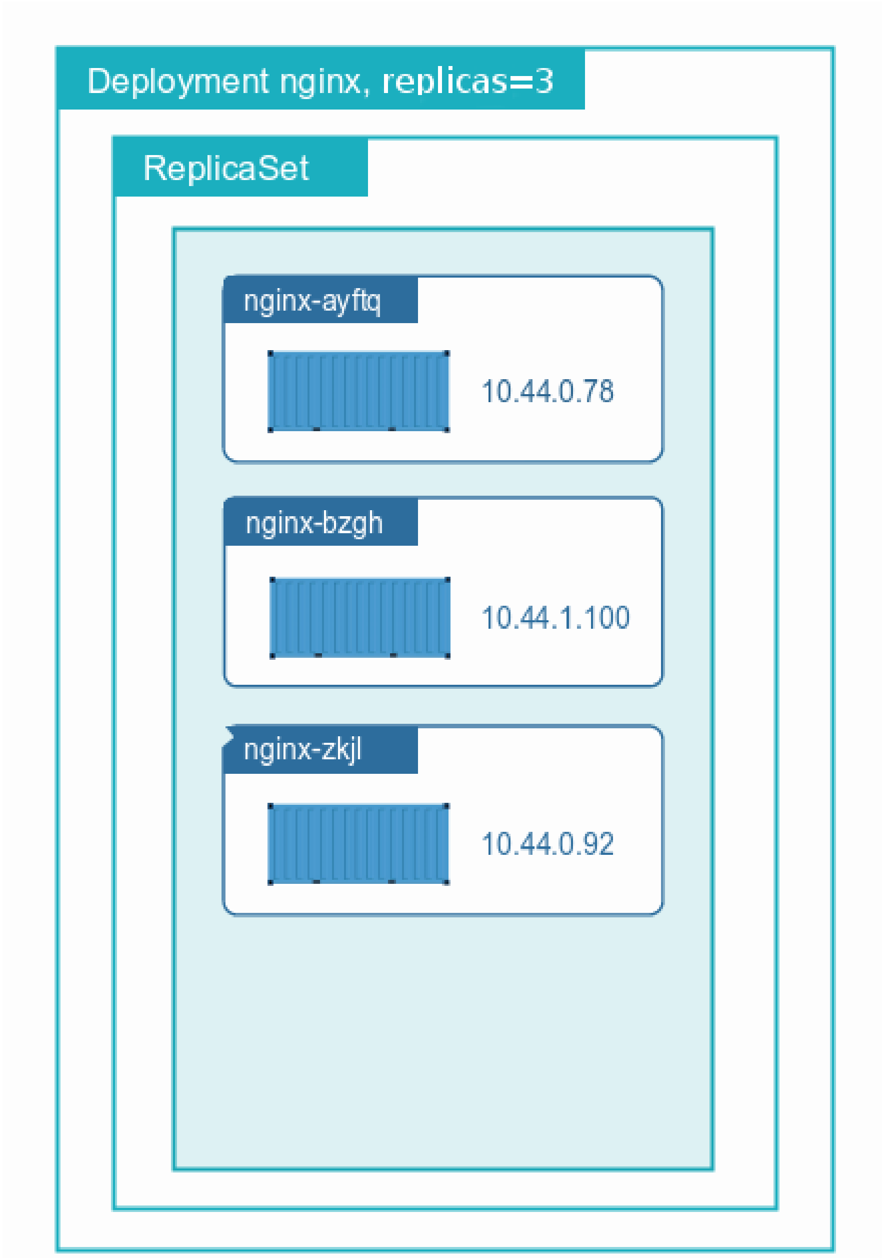
Deployments

- A Deployment provides declarative updates for Pods and ReplicaSets
- Deployment creates ReplicaSet, which creates the Pods
- Updating a deployment creates new ReplicaSet and updates the revision of the deployment.
- During update pods from the initial RS are scaled down, while pods from the new RS are scaled up.
- Rollback to an earlier revision, will update the revision of Deployment
- The --record flag of kubectl allows us to record current command in the annotations of the resources being created or updated
- Strategy – how to replace the old pods
  - Rolling update (default): maxUnavailable, maxSurge
  - Recreate

Working with Deployments

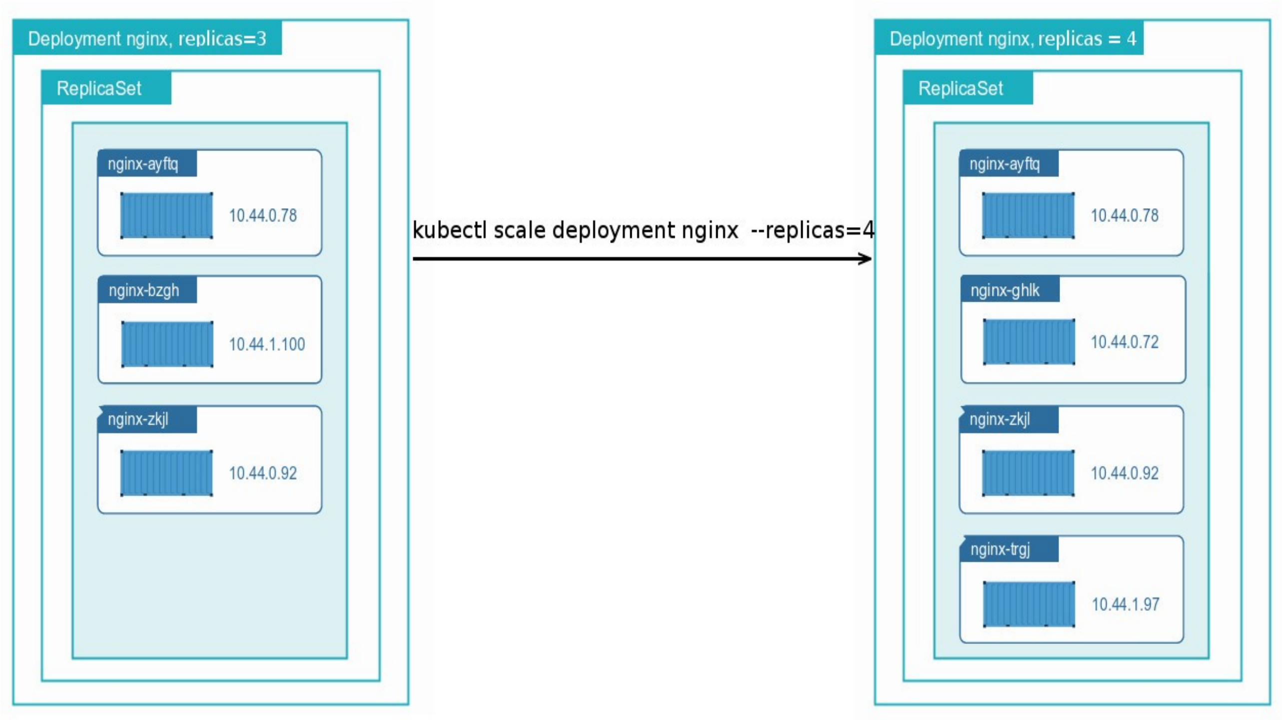
- Creating a deployment
  - kubectl run ghost --image=ghost --record
  - kubectl create -f dep1.yaml --record
  - dep1.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
```



Working with Deployments (cont)

- Check the status
  - kubectl get deployment nginx [--watch]
  - kubectl get deployment nginx -o yaml
  - kubectl describe deployment nginx
- Scale a deployment
  - kubectl scale deployment nginx --replicas=4



Working with Deployments (cont)

- Update a deployment
  - kubectl set image deployment/nginx nginx=nginx:1.7.9 --all=true
  - kubectl edit deployment nginx
- Check the status of a rollout
  - kubectl rollout status deployment nginx
  - kubectl rollout history deployment nginx
- Undo a rollout
  - kubectl rollout undo deployment/nginx [--to-revision=2]
- Pause and resume a deployment – allows multiple changes
  - kubectl rollout pause deployment/nginx
  - kubectl rollout resume deployment/nginx

Jobs, CronJobs

- A job creates one or more pods and ensures that a specified number of them successfully terminate.
- Jobs can be used to reliably run a Pod to completion the specified number of times (*.spec.completions*)
- Jobs can run multiple Pods in parallel (*.spec.parallelism*)
- Pods in a Job can only use *Never* or *OnFailure* as their RestartPolicy
- It is up to the user to delete old jobs after noting their status
- Deleting a Job will delete the related Pods
- If Pods are failing, the Job will create new Pods forever. The *.spec.activeDeadlineSeconds* will limit the time for which a Job will create new Pods.
- CronJobs can create Jobs once or repeatedly at specified times
- *.spec.jobTemplate* will specify the Job to be created
- concurrencyPolicy: *Allow*, *Forbid*, *Replace*

Jobs example

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  completions: 10
  parallelism: 3
  template:
    metadata:
      name: pi
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
      restartPolicy: Never
```

CronJobs example

```yaml
apiVersion: batch/v2alpha1
kind: CronJob
metadata:
  name: cron-pi
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
   spec:
    completions: 10
    parallelism: 3
    template:
     metadata:
       name: pi
     spec:
       containers:
       - name: pi
         image: perl
         command: ["perl",  "-Mbignum=bpi", "-wle", "print bpi(2000)"]
       restartPolicy: Never
```

DaemonSets

- A DaemonSet ensures that all (or some) nodes run a copy of a pod
- When nodes are added to the cluster, pods are added to them
- When nodes are removed from the cluster, those pods are garbage collected
- To run pods only on some nodes:
  - *.spec.template.spec.nodeSelector* – pods started on nodes that match the node selector
  - *.spec.template.spec.affinity* – pods are created on nodes that match the node affinity
- If node labels are changed, the DaemonSet will promptly adapt
- Deleting a DaemonSet will delete the pods (except –*cascade=false*)
- UpdateStrategy:
  - OnDelete - new pods will only be created when the old ones are manually deleted
  - RollingUpdate - after you update a DaemonSet template, old pods will be killed

Exercise 3: Kubernetes workloads

- Task 1: Working with pods
- Task 2: Working with deployments

4. Accessing the applications

- Services

Services

- Service – an abstraction which defines a logical set of Pods and a policy by which to access them
- The service maps an incoming port to a target port
- The pods targeted are defined by the selector → Endpoints
- We can have services without selector → no Endpoints object is created automatically
- iptables proxies depends on working readiness probes
- Service discovery:
  - Environment variables – are created when the pod is created → requires ordering (the service should be defined first)
  - DNS – optional cluster add-on. No ordering is required.

Service types

- ClusterIP: Exposes the service on a cluster-internal IP – only reachable from within the cluster. Default
- NodePort: Exposes the service on each Node’s IP at a static port. The service will be reachable from outside the cluster using NodeIP:NodePort
- LoadBalancer: Exposes the service externally using a cloud provider’s load balancer.
- ExternalName: Maps the service to the contents of the externalName field, by returning a CNAME record with its value.

Working with Services

- Expose the ports of a deployment/RC
  - kubectl expose deployment nginx --port=80 --type=NodePort
- Create services from file:

```
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80

kubectl create -f svc1.yaml
```

Working with Services

- Get service information:
  - kubectl get svc
  - kubectl describe svc
- Check service discovery
  - kubectl exec -ti busybox env
  - kubectl exec -ti busybox nslookup nginx
- Check the iptables rules on the nodes
  - iptables -t nat -L -n
  - iptables -L -n

Exercise 4: Services

- Working with services

5. Persistent storage in kubernetes

- Volumes
- Persistent volumes and volume claims
- Secrets
- ConfigMaps

Volumes

- By default the container filesystem is ephemeral – recreated each time when the container starts → a clean state each time → can be a problem for non trivial applications
- A pod can have multiple containers that are sharing files.
- A volume in the simplest form is just a directory which is accessible to the containers in a pod.
- The type of volume determines the backend for the directory.
- The pod definition specifies what volumes are provided (the *spec.volumes* field), and where are these mounted in the containers (the *spec.containers.volumeMounts* field).
- The containers are independently specifying where to mount each volume (the same volume can be mounted on different path in different containers).

Volume example

apiVersion: v1
kind: Pod
metadata:
  name: test-pd
spec:
  containers:
  - image: gcr.io/google_containers/test-webserver
    name: test-container
    volumeMounts:
    - mountPath: /cache
      name: **cache-volume**
  volumes:
  - name: **cache-volume**
    emptyDir: {}

Volume types

- Kubernetes supports several volume types:
  - emptyDir – initially empty; deleted when the pod is deleted (survives crashes)
  - hostPath – mounts a directory from the host into the pod. The content is host specific → pods with identical specs can behave differently on different nodes.
  - gcePersistentDisk – mounts a Google Compute Engine (GCE) Persistent Disk into the pod. Content preserved on pod delete → prepopulate, data “hand off”
  - awsElasticBlockStore - mounts an Amazon Web Services EBS Volume into the pod. Content preserved.
  - nfs – allows an existing NFS share to be mounted into the pod. Allows multiple writers. The server should be configured. Content is preserved.
  - iscsi – single writer. Can be mounted read only by multiple pods.
  - glusterfs – multiple writers.
  - rbd - single writer. Can be mounted read only by multiple pods.
  - cephfs – multiple writers.
  - secret
  - persistentVolumeClaim

Persistent Volumes

- PersistentVolume (PV) – a cluster resource that hides the details of storage implementation from the pod.
  - Can be of different types (HostPath, NFS, iSCSI, RBD, … plugins)
  - Are independent from the pods that are using them.
- PersistentVolumeClaim (PVC) – a request for storage by a pod.
  - PVCs will consume PV resources.
  - PVC can request size, access mode, storage class.
- StorageClass – describes the “classes” of storages
  - Classes can map to quality-of-service levels, backup policies, …
  - Allows for dynamic provisioning of Pvs.
- The pod definition will use the PVC for defining the volumes consumed by the containers.
- Dynamic provisioning is possible using the StorageClass definition.
  - A StorageClass will contain the *provisioner* and *parameter* fields.

Persistent Volume example

– First we define the PV:

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: nfs001
spec:
  capacity:
    storage: 10Gi
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  nfs:
    path: /tmp
    server: 10.10.10.1
```

Persistent Volume example (cont)

- We define the PVC (the claim):

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi
  storageClassName: slow
```

Persistent Volume example (cont)

- the Pod (the consumer):

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"
        name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim
```

Secrets

- Secret objects are intended to hold sensitive information, such as passwords.
- Safer than putting sensitive information into pod definition, or docker images.
- Secrets can be used by pods as files in a volume, or injected by the kubelet.
- Secrets can be created from files, or directly specifying them:
  - kubectl create secret generic mysql --from-literal=password=mypasswd
- Checking secrets:
  - kubectl get secret mysql -o yaml

Using Secrets as environmental variables

```
. . .
spec:
    containers:
    - image: mysql:5.5
      name: mysql
      env:
      - name: MYSQL_ROOT_PASSWORD
        valueFrom:
            secretKeyRef:
                name: mysql
                key: password
```

Using Secrets as volumes

```
...
spec:
  containers:
  - image: busybox
    command:
      - sleep
      - "3600"
    volumeMounts:
    - mountPath: /mysqlpassword
      name: mysql
    name: busy
  volumes:
  - name: mysql
      secret:
          secretName: mysql
```

- kubectl exec -ti busybox -- cat /mysqlpassword/password

ConfigMaps

- ConfigMap objects are intended for passing information that tends to be stored in a single config file
- Can store key-value pairs, or plain configuration files
  - kubectl create configmap special-config --from-literal=special.how=very
  - kubectl create configmap mymap –from-file=app.conf
- Check the values stored in the map
  - kubectl get configmap mymap -o yaml
- Passing values to pods:
  - As environmental variables (part of the pod definition):

```
env:
    - name: SPECIAL_LEVEL_KEY
      valueFrom:
        configMapKeyRef:
          name: special-config
          key: special.how
```

  - As volumes:

```
volumes:
  - name: config-volume
    configMap:
      name: special-config
```

Exercise 5: Storage in Kubernetes

– Use a volume in two containers