Workload

Namespaces and capabilities

Namespaces

- Namespaces are well suited for carving out environments within the cluster like development, staging, and production

- network policies, RBAC policies, and quotas

Namespaces - builtin

- **default**: Adding an object to a cluster without providing a namespace will place it within the default namespace. It cannot be deleted.
- The **kube-public** namespace is intended to be globally readable to all users with or without authentication.
- The **kube-system** namespace is used for Kubernetes components managed by Kubernetes. It is intended to be managed directly by the system and as such, it has fairly permissive policies.

Create a namespace

```
kubectl create namespace demo-namespace
```

```
# demo-namespace.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: demo-namespace
```

Limit resources in a namespace

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    pods: "4"
    requests.cpu: "1"
    requests.memory: 1Gi
    limits.cpu: "2"
    limits.memory: 2Gi

$ kubectl create -f ./resources.yaml --namespace=myspace
```

Admission controllers

- Kubernetes can be configured to use so admission plugins:
  `kube-apiserver --enable-admission-plugins=…`

- Runs after authorization but before storage of the object
- Intercepts calls and fills default values, checks settings and possibly authorization

- Default a set is enabled:
  `kube-apiserver -h | grep enable-admission-plugins`

https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/

Set the security context for a Pod

- You can change the userid & groupid of the user inside the containers.
- Set the context to ‘privileged’ to allow the process almost the same rights as it would have running outside Kubernetes (access to the host)
- Break down unrestricted root access by adding or revoking linux capabilities
  - Like binding to a port < 1024 as non root

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo
spec:
  securityContext:
    runAsUser: 1000
    runAsGroup: 3000
    fsGroup: 2000
  volumes:
  - name: sec-ctx-vol
    emptyDir: {}
  containers:
  - name: sec-ctx-demo
    image: busybox
    command: [ "sh", "-c", "sleep 1h" ]
    volumeMounts:
    - name: sec-ctx-vol
      mountPath: /data/demo
    securityContext:
      allowPrivilegeEscalation: false
```

Set the security context for a Container

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-2
spec:
  securityContext:
    runAsUser: 1000
  containers:
  - name: sec-ctx-demo-2
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      runAsUser: 2000
      allowPrivilegeEscalation: false
```

Capabilities

- Settings Linux capabilities in the container

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

Allow privileged containers

Add --allow-privileged=true to:

# kubelet config
sudo vim /var/snap/microk8s/current/args/kubelet

#kube-apiserver config
sudo vim /var/snap/microk8s/current/args/kube-apiserver
Restart services:

sudo systemctl restart snap.microk8s.daemon-kubelet.service
sudo systemctl restart snap.microk8s.daemon-apiserver.service

Kubernetes POD Security Policy (PSP)

- Implemented as Admission Controller
- Restrict the containers allowed to run on the cluster
  - Running of privileged containers
  - Allowed hostpaths
  - RunAs user
  - Allowed (and default) capabilities
  - Whitelist of Volume Types

Example PSP

```
allowedHostPaths:
  # This allows "/foo", "/foo/", "/foo/bar" etc., but
  # disallows "/fool", "/etc/foo" etc.
  # "/foo/../" is never valid.
  - pathPrefix: "/foo“
```

```
apiVersion: extensions/v1beta1
kind: PodSecurityPolicy
metadata:
  name: example
spec:
  privileged: true
  seLinux:
    rule: RunAsAny
  supplementalGroups:
    rule: RunAsAny
  runAsUser:
    rule: RunAsAny
  fsGroup:
    rule: 'MustRunAs'
    ranges:
      - min: 1
        max: 65535
  volumes:
  - '*'
```

```
$ kubectl create -f psp.yaml
podsecuritypolicy "example" created

$ kubectl get psp
NAME      PRIV   CAPS  SELINUX    RUNASUSER   FSGROUP   SUPGROUP  READONLYROOTFS   VOLUMES
example   true   []    RunAsAny   RunAsAny    MustRunAs RunAsAny  false            [*]
```

Kubernetes Networking Policies

- Control networking at POD level
- Like firewalling / iptables
- Using Kubernetes context like pod labels, namespaces, etc.
- Possible 3rd party plugins for pod overlay networks
  https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-backend-egress
  namespace: default
  spec:
    podSelector:
      matchLabels:
        tier: backend
        policyTypes:
          - Egress
          egress:
            - to:
              - podSelector:
                matchLabels:
                  tier: backend
```

```
ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
```

PID 1

- A process running as PID 1 inside a container is treated specially by Linux: it ignores any signal with the default action. So, the process will not terminate on SIGINT or SIGTERM unless it is coded to do so.

Liveness probe

- Simple process check is not enough
- Kubernetes can check if your application inside the container is alive by doing a liveness check.
- These need to be added manually into the manifest

```
livenessProbe:
  httpGet:
    path: /
    port: "80"
  initialDelaySeconds: 5
  timeoutSeconds: 1
  periodSeconds: 10
  failureTreshold: 3
  successThreshold: 1
```

Liveness probe

- HTTP type:
- Checks periodically by polling at the given path and portnumber
- Result should be statuscode between 200 and 400
- If it fails <failureTreshold> times the container will be restarted

Readiness probe

- Describes when a container is ready to handle user requests
- Configured just like liveness probes
- Combination of readiness and liveness probes makes sure only healthy containers are running
- Types
  - HTTP check
  - tcpSocket (tests if port can be opened)
  - Exec (runs a command, 0 resultcode is success)

Limits

- Resource limits
- Default limits
- Eviction

Resource management

- In addition to simplified distributed application deployment we want to increase overall utilization of our compute nodes
- Basic cost of operating a machine is constant, regardless of being idle or fully loaded
- With Kubernetes managing resource packing we can go up well over 50%
- To do this Kubernetes needs to know what can be expected from your application in terms of resource requirements, so packing can be optimal across the cluster

Resource management

- Based on 2 different resource metrics
  - Requested resources
    - Minimum required:
    - CPU (i.e. 500m is half a cpu)
    - Memory (i.e. 128Mi is 128MB of memory)
    - Expressed in container section(s) of manifest file
  - Resource limits
    - Same settings, but maximizing/capping to the configured level

Resource management

```
containers:
  - name: my-pod
    image: nginx
    resources:
        requests:
          cpu: 100m
          memory: 128Mi
        limits:
          cpu: 500m
          memory: 256Mi
```

The downward API

- There are two ways to expose Pod and Container fields to a running Container:

- Environment variables
- DownwardAPIVolumeFiles

```
volumeMounts:
        - name: podinfo
          mountPath: /etc/podinfo
          readOnly: false
  volumes:
    - name: podinfo
      downwardAPI:
        items:
          - path: "labels"
            fieldRef:
              fieldPath: metadata.labels
          - path: "annotations"
            fieldRef:
              fieldPath: metadata.annotations
```

Using hooks and initContainers

- Specialized containers that run before app containers in a Pod.
- Init containers can contain utilities or setup scripts not present in an app image.
- Init containers are exactly like regular containers, except:
  - Init containers always run to completion.
  - Each init container must complete successfully before the next one starts.

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice;
sleep 2; done;']
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2;
done;']
```

```
apiVersion: v1
kind: Service
metadata:
  name: myservice
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9376
---
apiVersion: v1
kind: Service
metadata:
  name: mydb
spec:
  ports:
  - protocol: TCP
    port: 80
    targetPort: 9377
```

- kubectl apply -f myapp.yaml
- kubectl describe -f myapp.yaml

- kubectl logs myapp-pod -c init-myservice # Inspect the first init container
- kubectl logs myapp-pod -c init-mydb # Inspect the second init container
- kubectl apply -f services.yaml

- What are Lifecycle hooks
- Kubernetes supports container lifecycle hooks which allows a handler code/script to be executed when Container lifecycle event occurs.
- There are two lifecycle hooks. PostStart hook executes immediately after container is created. PreStop hook is called just before the container is terminated. Both the hooks take no parameters.
- K8s support two types of handlers for the lifecycle hooks. Exec handler is used for calling shell command/script while HTTP handler is used for calling HTTP endpoint

Lifecycle hook example

```
apiVersion: v1
kind: Pod
metadata:
  name: lifecycle-demo
spec:
  containers:
  - name: lifecycle-demo-container
    image: nginx
    lifecycle:
      preStop:
        exec:
          command: ["/usr/sbin/nginx","-s","quit"]
```

Init Containers	Lifecycle Hooks
Configured at Pod Level. All init Containers execute before app Container	Separate hooks for each container in a Pod.
Don’t support post Stop lifecycle event for Pod	Supports both preStart and postStop hook
They are just like containers with own image, volumes and resources	They are executed inside parent container as script or HTTP command
Init Containers are guaranteed to execute before Pod is initialized.	There is no guarantee that the preStart hook will execute before the container ENTRYPOINT

Usage scenarios- Init Containers

- Delay or Block the startup of app Containers until some set of preconditions are met. E.g . sleep 60
- In some cases if one Pod has startup dependency on another (Avoid as much as possible)
- For multi-container Pod, perform common task required at startup

Usage scenarios – lifecycle hooks

- Perform cleanup before container is terminated with preStop hook
- Save state of container before termination with preStop hook
- Perform post configuration tasks on application startup inside container.

Stateful sets

- Manages the deployment and scaling of a set of Pods
- Provides guarantee about the ordering and uniqueness of these Pods.
- Persistent Volume per pod (<pod>-01, <pod>-02 etc)

Pod priority and preemption

- Pods can have priority
- Relative to other pods
- If a pod cannot be scheduled it preempts lower priority pods

- Add one or more PriorityClasses
- Use `priorityClassName` in Pod definition

PriorityClass example

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority
value: 1000000
globalDefault: false
description: "This priority class should be used for XYZ service pods only."
```

PriorityClass with disabled preemption

```
apiVersion: scheduling.k8s.io/v1
kind: PriorityClass
metadata:
  name: high-priority-nonpreempting
value: 1000000
preemptionPolicy: Never
globalDefault: false
description: "This priority class will not cause other pods to be preempted."
```

Using PriorityClass

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  priorityClassName: high-priority
```

Disable preemption

```
apiVersion: kubescheduler.config.k8s.io/v1alpha1
kind: KubeSchedulerConfiguration
algorithmSource:
  provider: DefaultProvider

...

disablePreemption: true
```

Eviction

- When resources become low can stop Pods to free up memory or storage
- Configure by Eviction Policy
  - Eviction trigger signal
  - Treshold value

Eviction setting

- Either hard or softeviction
- Set this in the kubelet configuration

```
--eviction-
hard="memory.available<100Mi,nodefs.available<1Gi,imagefs.available<1Gi"


--eviction-soft="memory.available<1.5Gi,…"
--eviction-soft-grace-period="memory.available=1m30s,…"
```

Taints

- Affinity rules set pods to requested nodes
- Taints are the opposite: do not allow (certain) pods on a node

- Example:
  `kubectl taint nodes node1 key=value:NoSchedule`
- Node 1 will not allow pods unless it has a matching toleration

Toleration

```
# podspec
tolerations:
- key: key
  operator: Equal
  value: “value”
  effect: NoSchedule
```

Affinity

```
#podspec
affinity:
  nodeAffinity:
    requiredDuringSchedulingIgnoredDuringExecution:
      nodeSelectorTerms:
      - matchExpressions:
        - key: kubernetes.io/hostname
          operator: In
          values:
          - nod1
```