|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Application Design and Build | | | | | | |
| Image Management | FROM node:x.y.z-alpine  COPY . /src  RUN npm install  EXPOSE 8080  ENTRYPOINT [“node”, “./app.js”] | | | | Base image  Copy app code into image at /src  Install dependencies  Application port  Command to run app | |
| Dockerfile -> Docker/ other build tools-> (OCI) Image (Open Containers Initiative) | | | | | |
| Managing Images w Docker | *docker build -t <tag> .*  *docker image ls*  *docker image pull <image name>*  *docker save <image name> --output test.tar*  *docker save <image> | gzip > image.tar.gz*  *docker commit <container> <new image name>*  *docker image tag <old tag> <new tag>*  *docker image rm <image name>*  *docker <command> --help* | | | | Save image as TAR file  Save as gzip  Save running container as new image  Re-tag image  Delete image | |
| Managing Images w Other Tools | 1) buildah: Image management  - Creates and manages OCI images. Only does images. – Don’t need a Dockerfile  *- buildah images. – buildah build -t <image name>*  *- buildah tag <image name> <new tag>.*  2) podman: Image and container management  - Creates & manages OCI images. Manages entire container lifecycle. Supports all Docker CLI commands. – Daemonless. – Uses buildah for builds  *- podman image ls. – podman build -t <image name> .*  *- podman image tag <image name> <new tag>. – podman save <image name> --output image.tar* | | | | | |
| Jobs and CronJobs | Jobs: run set num of Pods to completion. – Managed by a controller. – Can run Pods in parallel  Job controller brings: restarts, retries, kill long-running, clean-up  CronJobs: run Jobs on a schedule | | | | | |
| Jobs | apiVersion: batch/v1  kind: Job  metadata:  name: ckad1  spec:  activeDeadlineSeconds: 10  ttlSecondsAfterFinished: 120  completions: 5  parallelism: 1  backoffLimit: 4  template:  spec:  restartPolicy: Never  containers:  - name: ctr  image: alpine:latest  command: [“echo”, “HELLO”] | | | Kill pod if still running after 10s  Clean up/Delete job 120s after finish/failed  Run 5 Pods  Run 1 at a time  Num of retries = 4 (exponential delay, 10s, 20s, … 6min)  [Never, OnFailure]. OnFailure = container re-run on same pod. Never = recreate Pod and run | | |
| CronJobs | apiVersion: batch/v1  kind: CronJob  metadata:  name: ckad-cron1  spec:  schedule: "\* \* \* \* \*"  startingDeadlineSeconds: 30  concurrencyPolicy: Allow  successfulJobsHistoryLimit: 5  failedJobsHistoryLimit: 2  jobTemplate:  spec:  … same as Job | | CronJobs check every 10s for new tasks  If 100+ tasks are missed, controller won’t try again  Timezone comes from k8s API server  [Minutes, Hour, Day of month, Month, Day of week]  Deadline for when Job is started, if missed, don’t start  If new Jobs can start, when previous one still running [Allow, Forbid, Replace]  How many history to keep | | | |
|  | *kubectl config set-context --current --namespace=<namespace>* | | | | | |
| Multi-Container Pods | Separation of concerns: every container only does 1 thing  Architecture: 1 main container + ≥ 0 sidecar containers  1) Ambassador pattern: sidecar/ambassador container is always running when main is running  - Offload communication responsibilities to ambassador (DB connection, routing, load balancing)  2) Adapter pattern: sidecar/adapter long-lived container that runs alongside main  - Transform data format/protocols btw main and other systems | | | | | |
| 3) Init containers: runs before main container  - Initialize stuff  Can run multiple init containers, but they run in the order it is listed  Main container only start running once all init containers are successful | | | | spec:  initContainers:  - name: svc-check  image: busybox  containers:  - name: main | |
| All containers in a Pod can access and share the same memory, volume, network, … | | | | | |
| Persistent & Ephemeral Volumes | Cloud K8s clusters can only use cloud storage. On-prem cluster use on-prem storage  Volume = storage for data persistence or shared storage  1) Use StorageClass to define a class of storage from an external storage system  2) Define Pods that reference a PersistentVolumeClaim that makes a reference to StorageClass  3) When Pod is scheduled, storage is dynamically provisioned and available in K8s cluster as a PersistentVolume  PV: piece of storage in cluster that has been provisioned by an admin or dynamically provisioned using Storage Classes  PVC: request for storage by user | | | | | |
| Ephemeral Volumes: emptyDir (temp files), configMap, secret | | | | | |
| Reclaim Policy: [Delete: delete volume when not bound to a Pod; Retain: won’t delete volume]  VolumeBindingMode: [Immediate: provision volume as soon as PVC created;  WaitForFirstConsumer: even if PVC is created, don’t create actual volume until a Pod claims it (so that volume is guaranteed to be created in same zone or region as pod)] | | | | | |
| apiVersion: storage.k8s.io/v1  kind: StorageClass  metadata:  name: sc  provisioner: linodebs.csi.linode.com  reclaimPolicy: Retain  allowVolumeExpansion: true  volumeBindingMode: WaitForFirstConsumer | | | | apiVersion: v1  kind: PersistentVolumeClaim  metadata:  name: pvc-wait  spec:  accessModes:  - ReadWriteOnce  resources:  requests:  storage: 10Gi  storageClassName: sc | |
| apiVersion: v1  kind: Pod  spec:  containers:  - name: app1’  image: app1-image  volumeMounts:  - name: temp-storage  mountPath: /data  volumes:  - name: temp-storage  emptyDir: {} | apiVersion: v1  kind: ConfigMap  metadata:  name: app-config  data:  APP\_MODE: production  LOG\_LEVEL: debug | | | | apiVersion: v1  kind: Secret  metadata:  name: db-password  data:  password: cGFzc…  # password encoded in base64 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Application Deployment | | | | | | | | | | | |
| Understanding Deployments | | | Deployments -> ReplicaSet -> Pod -> Container  Deployment Options: 1) Rolling Update. 2) Blue/Green. 3) Canary Deployment. 4) Rollbacks | | | | | | | | |
| apiVersion: apps/v1  kind: Deployment  metadata:  name: frontend  labels:  tier: frontend  spec:  selector:  matchLabels:  tier: frontend  template:  metadata:  labels:  tier: frontend  spec:  containers:  - name: nginx  Image: nginx:alpine | | | | | | Selector used to “select” template to use (based on labels)  Template to use to create the Pod/container | | |
| Declarative: YAML file  Imperative: *kubectl create deployment nginx --image=nginx:alpine --dry-run=client -o yaml > deploy.yaml* # create deployment and save to YAML file  *kubectl create -f deploy.yaml* # Create deployment  *kubectl scale deployment nginx --replicas=4* # Scale deployment pods to 4  *kubectl set image deployment/nginx nginx* # Change deployment image | | | | | | | | |
| Blue/Green Deployment | | Mirrors production env to check deployment before it’s publicly available  Runs 2 identical production env at the same time. New (green). Old (blue)  Traffic routed from blue to green when checks pass  Blue/Green Deployment need 3 main K8s resource: 1) Test Service (internal testing). 2) Public Service. 3) Deployment | | | | | | | | | |
| kind: Service  apiVersion: v1  metadata:  name: blue-test-service  labels:  env: blue  spec:  type: LoadBalancer  selector:  app: nginx  role: blue  ports:  - port: 9000  targetPort: 80 | | | kind: Service  apiVersion: v1  metadata:  name: public-service  labels:  env: prod  spec:  type: LoadBalancer  selector:  app: nginx  role: blue  ports:  - port: 80  targetPort: 80 | | | | | | apiVersion: apps/v1  kind: Deployment  metadata:  name: blue-deployment  spec:  replicas: 2  selector:  matchLabels:  app: nginx  role: blue  template:  metadata:  labels:  app: nginx  role: blue  spec:  containers:  - name: blue |
| *kubectl set selector svc [service-name] ‘role=green’* # Change from blue to green deployment | | | | | | | | | |
| Canary Deployment | | Service will route small percentage of production traffic to canary version  3 main K8s resource: 1) Service. 2) Stable Deployment. 3) Canary Deployment  - Use number of replicas to control % of traffic as Service Load balancer will split traffic | | | | | | | | | |
| kind: Service  apiVersion: v1  metadata:  name: stable -service  labels:  app: customer-app  spec:  type: LoadBalancer  selector:  app: customer-app  ports:  - port: 9000  targetPort: 80 | | apiVersion: apps/v1  kind: Deployment  metadata:  name: stable-deployment  spec:  replicas: 4  selector:  matchLabels:  app: customer-app  track: stable  template:  metadata:  labels:  app: customer-app  track: stable  spec:  containers:  - name: stable-app  image: v1 | | | | | | apiVersion: apps/v1  kind: Deployment  metadata:  name: canary-deployment  spec:  replicas: 1  selector:  matchLabels:  app: customer-app  role: canary  template:  metadata:  labels:  app: customer-app  role: canary  spec:  containers:  - name: canary-app  image: v2 | |
| *kubectl create deployment …*  *kubectl create service … OR kubectl expose deployment …*  *kubectl run -it --restart=Never --image=alpine temp-pod* # Create pod and shell into it | | | | | | | | | |
| Rolling Update Deployment | | – Zero downtime deployment. - Increase new Pods while decreasing old Pods  - Service handles load balancing traffic to available Pods  - New Pods only scheduled on available Nodes  Deployments support 2 strategy: rolling update (default) OR recreate (can have down-time) | | | | | | | | | |
| kind: Deployment  spec:  replicas: 4  minReadySeconds: 1  progressDeadlineSeconds: 60  revisionHistoryLimit: 5  strategy:  type: RollingUpdate  rollingUpdate:  maxSurge: 1  maxUnavailable: 1 | | | | Seconds to wait to consider new Pod as healthy  Seconds to wait before reporting stalled Deployment  Number of ReplicaSets that can be rolled back  RollingUpdate or Recreate  How many Pods can be added above the replicas count  How many of existing Pods can be unavailable | | | | | |
| Need save config to enable rollback  *kubectl create -f deployment.yaml --save-config* # Save config in resource’s annotation  *kubectl apply -f deployment.yaml --record=true* # Save config  *kubectl annotate deployment [name] kubernetes.io/change-cause=”Change details” --overwrite=true*  *kubectl get deploy [name] -o json | jq ‘.metadata.annotations.”kubernetes.io/change-cause”’*  *kubectl rollout statue deployment [deployment-name]* # Get info about deployment | | | | | | | | | |
| Rollback Deployment | | *kubectl rollout history deployment [deployment-name]* # View history of a deployment  *kubectl rollout history deployment [name] --revision=2* # View history of a specific deployment  *kubectl rollout undo -f deployment.yaml --to-revision=2* # Rollback to specific revision | | | | | | | | | |
| Helm Package Manager | Simplifies deploying and managing K8s apps using charts (pre-configured K8s resource definitions)  - Package manager for K8s. – Use charts to install, upgrade, uninstall K8s apps  - Helm Client – CLI client for end users. – Helm Library: logic for executing operations | | | | | | | | | | |
| Chart: bundle of info used to create an instance of a K8s app  Config: config info that can be merged into a packaged chart to create a releasable object  Release: running instance of a chart (combined w a specific config) in K8s | | | | | | | | | | |
| *helm -h*  *helm search hub*  *help repo add*  *helm repo update*  *helm search repo*  *helm show values*  *helm pull --untar*  *helm install*  *helm upgrade*  *helm status*  *helm list*  *helm uninstall* | | | | | | Help documentation  Search for charts on defaults Artifact Hub  Adds a chart repo to local Helm setup  Update all repos  Searches for charts in repo added locally  Shows default values of a chart  See default values of chart in their respective files  Installs a Helm chart into K8s cluster as a release  Upgrade a release to new version  Show current status of a release  List all Helm release currently installed in cluster  Deletes a Helm release | | | | |
| Kustomize | Manage and simplifies K8s configs.  – Allows overriding elements of manifest files for env like Dev, QA, Prod using patching & overlays  *kubectl apply -k <kustomize directory>*  *kubectl kustomize* . # See yaml file w customizations before deploy | | | | | | | | | | |
| resources:  - deployment.yaml  - service.yaml  commonLabels:  app: mykustomapp  commonAnnotations:  app: mykustom-annotations  namePrefix:  kustom-  nameSuffix:  -v1  configMapGenerator:  - name: mykustom-map  env: config.properties | | | | | | | Base kustomization.yaml  Resources: files that Kustomize should import  Common Attributes: common configs in manifests like labels, annotations, names  Generators: manages complexities w Creating/managing ConfigMaps and Secrets | | | |
| - base  - deployment.yaml  - kustomization.yaml  - service.yaml  - overlays  - dev  - config.properties  - kustomization.yaml  - replicas.yaml  - prod  - config.properties  - kustomization.yaml  - replicas.yaml | | | | | | | Base folder contains manifest files and an additional kustomization.yaml  Overlays folder contains env specific config | | | |
| bases:  - ../../base  namespace: prod  patches:  - replicas.yaml  configMapGenerator:  - name: mykustom-map  Env: config.properties | | | | | | | Overlays/<env>/kustomization.yaml  Patching: used to apply diff customizations to resources | | | |
|  |  | | | | | | | | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Application Observability and Maintenance | | | | | |
| API Deprecations | | Admission controller: code that intercepts request to K8s API server prior to persistence of the obj, but after the request is authenticated and authorized  User issue kubectl command -> Authentication -> Authorization -> Admission Controller -> etcd store + Pods  - Customize what can run on a cluster. – May be validating, mutating, or both  - Limit requests to create, delete, modify objs. – Do not (and cannot) block requests to read  - Enforce security policies. – Block insecure images from running.  – Compiled into the kube-apiserver binary  - Can see in /etc/Kubernetes/manifests/kube-apiserver.yaml  -- something like enable-admission-plugins=NodeRestriction,NamespaceAutoProvision  - Or to view: *kubectl describe pod kube-apiserver -n kube-system*  - *kube-apiserver -h | grep enable-admission-plugins* | | | |
| 1) LimitRanger: apply default Pod memory/cpu limits for a namespace  2) PersistentVolumeClaimResize: check incoming PVC resize requests. Prevents resizing of all claims by default  3) NamespaceAutoProvision: examines requests and checks if reference namespace exists. Create if not found | | | |
| *kube-apiserver --enable-admission-plugins=NamespaceAutoProvision,LimitRanger*  *kube-apiserver --disable-admission-plugins=NamespaceAutoProvision* | | | |
| *kubectl version -o yaml*  *kubectl api-resources --sort-by=name* # View admission controller plugins for kube-apiserver  An API can be used to extend the K8s API and is used in the REST path as well as the apiVersion field of a resource  Core Group will have apiVersion: v1  Named Groups will have something like apiVersion: batch/v1  So all apis -> core group OR apps OR batch or certificates.k8s.io, …  In each group, can have API versions: alpha -> beta -> stable/GA (generally available) | | | |
| K8s deprecation rules:  1) API elements may only be removed by incrementing the version of the API group  2) API objs must round-trip btw API versions w/o info loss  3) An API version in each track may not be deprecated in favor of a less stable API version  4a) API lifetime is determined by the API stability level  - GA API versions may be marked as deprecated, but must not be removed within a major version of K8s. - Beta … - Alpha may be removed in any release w/o prior deprecation notice  4b) The preferred API version and storage version for a group may not advance until after a release has been made that supports both the new and previous versions | | | |
| *kubectl explain deploy* # View API group and version info for Deployments  To enable alpha feature, need go into /etc/Kubernetes/manifests/kube-apiserver.yaml file,  --runtime-config=batch/v2alpha1  # Get the preferred version for the certificates.k8s.io API group  *kubectl proxy 8001*  *curl localhost:8001/apis/certificates.k8s.io* | | | |
| Probes & Health Checks | Probe: diagnostic performed periodically by kubelet on container  1) Readiness Probe: determine if a Pod should receive requests  - If failed, stop sending traffic (maybe app starting up or temporarily unavailable)  2) Liveness Probe: determine if a Pod is healthy and running as expected  - Failed Pod containers are recreated by default (restartPolicy defaults to Always)  3) Startup Probe: use if container takes very long to start up (legacy) | | | | |
| Probe Types:  1) ExecAction: executes an action inside the container  2) TCPSocketAction: TCP check against the container’s IP address on a specified port  3) HTTPGetAction: HTTP GET request against container  - Probes can have results: Success, Failure, Unknown | | | | |
| apiVersion: v1  kind: Pod  spec:  containers:  - name: my-nginx  image: nginx: alpine  readinessProbe:  httpGet:  path: /  port: 80  initialDelaySeconds: 2  periodSeconds: 5 | | Check / on port 80  Wait 2 seconds (default 0)  Check every 5 seconds (default 10) | | |
| spec:  containers:  - name: my-nginx  image: nginx: alpine  livenessProbe:  httpGet:  path: /  port: 80  initialDelaySeconds: 15  timeoutSeconds: 2  periodSeconds: 5  failureThreshold: 1 | | Timeout after 2 seconds  Allow 1 failure before marking Pod as failed | | |
| spec:  containers:  - name: liveness  image: k8s.gcr.io/busybox  args:  - /bin/sh  - -c  - touch /tmp/healthy; sleep 30; rm -rf /tmp/healthy; sleep 60;  livenessProbe:  exec:  command:  - cat  - /tmp/healthy  port: 80  initialDelaySeconds: 5  periodSeconds: 5 | | | | Define action/command to execute |
| Tools to monitor K8s apps | Key monitoring and alerting options: - Web UI Dashboard, Metrics Server, kube-state-metrics, Prometheus, Grafana, …  Metrics Server: collects resource metrics from Kubelets and exposes them in K8s apiserver through Metrics API for use by Horizontal Pod Autoscaler and Vertical Pod Autoscaler  - Metrics API can also be accessed by kubectl top  Kubelet – Summary API -> Metrics Server – Metrics API -> API Server -> kubectl top | | | | |
| *kubectl top <node OR pod>*  *kubectl get --raw /api/v1/nodes/…* | | | | |
| Container Logs | *kubectl get pod <pod name> -o yaml*  *kubectl describe pod <pod name>*  *kubectl exec <pod name> -it sh*  *kubectl logs <pod name>*  *kubectl logs <pod name> -c <container name>*  *kubectl logs deployment/<deployment name>*  *kubectl logs -p <pod name>*  *kubectl logs -f <pod name>*  *kubectl logs --tail=20 <pod name>*  *kubectl logs --since=10s <pod name>*  *kubectl logs -l app=backend --all-containers=true*  *kubectl get pods counter -o jsonpath=’{.status.containerStatuses[\*].ready}’* | | | Show logs from previously terminated Pod  Stream Pod logs to console  View specific num of log lines  Return log entries newer than (5s, 2m, 1h,…)  Return log entries for Pods w specific labels  List ready state for each of the pod containers | |
| Debugging K8s | *kubectl describe pod <pod name>*  *kubectl get pod <pod name> -o yaml*  *kubectl get deployment <name> -o wide*  *kubectl get events --field-selector type=warning --all-namespaces*  *kubectl exec <pod name> -it – sh*  *kubectl debug myapp -it --copy-to=myapp-debug --container=myapp -- sh* | | Get specific Pod events  Shell into a Pod container  Create a copy of Pod w debugging utilities and change command | | |
|  |  | | | | |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Application Environment, Configuration and Security | | | | | | | | |
| Resources that extend K8s | Custom resource: extend K8s by creating new object types  - Need to create own API that K8s will use to manipulate its custom resources  - Need create custom operators | | | | | | | |
| Custom Resource Definitions (CRD): define custom resource | | | | | | | |
| apiVersion: apiextensions.k8s.io/v1  kind: CustomResourceDefinition  metadata:  name: customs.example.com  spec:  group: example.com  scope: Namespaced  names:  plural: customs  singular: custom  kind: Custom  shortNames:  - cs | | | | | versions:  - name: v1  served: true  storage: true  schema:  openAPIV3Schema:  type: object  properties:  spec:  type: object  properties:  field:  type: object  properties:  subfield1:  type: string  subfield2:  type: integer | | |
| apiVersion: example.com/v1  kind: Custom  metadata:  name: custom-object  spec:  field:  subfield1: random:string  subfield2: 4 | | | | | To use the custom resource defined above | | |
| Authentication, Authorization, Admission Control | | | Authentication: check identity. Authorization: what permissions you have | | | | | |
| Access Control: Role-Based Access Control (RBAC) OR Attribute-Based Access Control (ABAC)  For RBAC: create roles, then assign roles to users  For role binding across all namespace, use ClusterRoleBinding with ClusterRole | | | | | |
| apiVersion: rbac.authorization.k8s.io/v1  kind: Role  metadata:  name: pod-viewer  rules:  - apiGroups: [“”]  resources: [“pods”, “nodes”]  verbs: [“list”] | | | | apiVersion: rbac.authorization.k8s.io/v1  kind: RoleBinding  metadata:  name: pod-viewer-binding  subjects:  - kind: User #User, Service or Group  name: username  namespace: default  roleRef:  - kind: Role  name: pod-viewer-role  apiGroup: rbac.authorization.k8s.io | |
| User -> Authentication -> Authorization -> Admission Control -> CRUD K8s resources  Admission Control: validate user request or mutate it to meet certain criteria or reject request  - Admission Controller details in /etc/kubernetes/manifests/kube-apiserver.yaml | | | | | |
| Requests, Limits, Quotas | During scheduling, Nodes are filtered to see if they meet parameters for Pod  - Nodes are then ranked based on which has the most resources and Pod affinity | | | | | | | |
| Pods Requests: estimated resources needed (CPU and memory)  Pods Limits: terminate resources when reached (not the Node being limited)  Limits must be higher than request, else pod won’t be deployed | | | | | | | |
| Resource Quotas: limits on namespace level  - Can also limit number of any kind of obj created | | | | | apiVersion: v1  kind: ResourceQuota  metadata:  name: ps-quota  namespace: some-namespace  spec:  hard:  requests.cpu: “10”  requests.memory: “20Gi”  limits.cpu: “20”  limits.memory: “40Gi”  pods: “10”  scopes:  - NotTerminating  - NotBestEffot | | |
| ConfigMaps | | Key value pairs, dictionary  - 1MB data limit  Ways of using ConfigMaps:  1) As an argument  2) As an env variable  3) As a file  4) With self-written code | | | apiVersion: v1  kind: ConfigMap  metadata:  name: dev-env-config  data:  database\_url: “jdbc:mysql://dev.example.com:3306/app”  app\_port: “8080”  v2\_ui: “true”  init.sh: |  #! /bin/bash  <file contents…> | | | |
| kind: Pod  spec:  containers:  - name: busybox  args: [“$(APP\_PORT)”]  env:  - name: DATABASE\_URL  valueFrom:  configMapKeyRef:  name: dev-env-config  key: database\_url  - name: APP\_PORT  valueFrom:  configMapKeyRef:  name: dev-env-config  key: app\_port | | volumes:  - name: init-volume  configMap:  name: dev-env-config  items:  - key: init.sh  path: /tmp/init.sh | | | | # Use this if arg name same as name defined in config map  args: [“$(APP\_PORT)”]  envFrom:  - configMapRef:  name: dev-env-config |
| Secrets | Similar to ConfigMaps but for sensitive encrypted data  Ways of using Secrets:  1) As an argument.  2) As an env variable.  3) As a file.  4) With self-written code | | | | | apiVersion: v1  kind: Secret  metadata:  name: db-pass-secret  type: Opaque  data:  db-pass: “dmFsdWutM=” | | |
| kind: Pod  spec:  containers:  - name: busybox  env:  - name: DB\_PASS  valueFrom:  secretKeyRef:  name: db-pass-secret  key: db-pass | | | | | kind: Pod  spec:  volumes:  - name: secrets-volume  secret:  secretName: db-pass-secret  containers:  - name: busybox  volumeMounts:  - name: secrets-volume  readOnly: true  mountPath: “/tmp/dp-pass” | | |
| Service Accounts | Allow a process in a container to access our K8s API (similar to a user)  - Can do RBAC. – Use tokens  - Need to bind service account to both role and pod | | | | | apiVersion: v1  kind: ServiceAccount  metadata:  name: reader-service  automountServiceAccountToken: true | | |
| apiVersion: rbad.authorization.k8s.io/v1  kind: RoleBinding  metadata:  name: pod-viewer-binding  subjects:  - kind: Service  name: reader-service  namespace: default  roleRef:  - kind: Role  name: pod-viewer-role  apiGroup: rbac.authorization.k8s.io | | | | | apiVersion: v1  kind: Pod  metadata:  name: web-pod  spec:  serviceAccountName: reader-service  containers:  - name: busybox | | |
| Security Contexts | Provide additional privilege and access control settings. Security Context Options:  - UID/GUID settings. – Privilege escalation. – Filesystem settings  - AppArmor/SELinux options. – Seccomp settings. – Linux capabilities | | | | | kind: Pod  spec:  securityContext:  runAsUser: 1000  runAsGroup: 3000  fsGroup: 2000 | | |
|  |  | | | | | | | |

|  |  |  |  |
| --- | --- | --- | --- |
| Services and Networking | | | |
| Network-Policies | Network that pods use: Pod network (implemented by 3rd party plugin). Default allow all  Network Policy: define ingress and egress . – Network Policies require plugins  - Traffic not in a policy is implicitly denied. – Can combine policies. – All rules are allow rules | | |
| apiVersion: networking.k8s.io/v1  kind: NetworkPolicy  metadata:  name: ps-ckad  spec:  podSelector:  matchLabels:  project: ckad  policyTypes:  - Egress  egress:  - podSelector:  matchLabels:  role: front-end  ports:  - protocol: TCP  port: 9000 | | For pod with this label  Allow out bound traffic to this pod |
| Services | Service sits in front of pods. Pod might get kill, create new one, but service name and IP will stay the same. – Service name is auto recorded in cluster’s internal DNS which all pods can access  - All pods will have /etc/resolv.conf which have internal IP of DNS  - Use Selector to select pods which Service will apply to | | |
| 1) ClusterIP: internal IP and port on the Pod network (internal)  2) NodePort: static port on all cluster nodes (external)  3) LoadBalancer: integrate w external cloud load-balancer | | |
| apiVersion: v1  kind: Service  metadata:  name: cluster-ip  spec:  type: ClusterIP  ports:  - port: 9000  targetPort: 8080  selector:  course: ckad  ipFamilyPolicy: PreferDualStack | apiVersion: v1  kind: Service  metadata:  name: node-port  spec:  type: NodePort  ports:  - nodePort: 30001  port: 9000  targetPort: 8080  selector:  course: ckad  ipFamilyPolicy: PreferDualStack | |
| Ingress | NodePort: need to know IP of nodes, and ports have to be btw 30000 and 32767  LoadBalancer: 1 to 1 mapping btw LB and Services/app  Ingress: allow 1 Load Balancer to map to multiple Services (only HTTP/HTTPS)  - Use host or path-based routing to send traffic to services on your K8s cluster  Client -> External Load Balancer -> Ingress -> Service (ClusterIP) -> Pod | | apiVersion: networking.k8s.io/v1  kind: Ingress  metadata:  name: ckad  annotations:  nginx.ingress.kubernetes.io/rewrite-tarrget: /  spec:  ingressClassName: nginx  rules:  - host: [www.app.com](http://www.app.com)  http:  paths:  - path: /  pathType: Prefix  backend:  service:  name: frontend  port:  number: 80  - host: dev.app.com  - host: app.com |
| In Ingress: 1) Ingress spec: defines rules. 2) Ingress Controller: implement rules  - Need to install 3rd party controller.  - No native one | |
| ingressclass: Run more than 1 ingress controller in a cluster | |
|  |  | | |