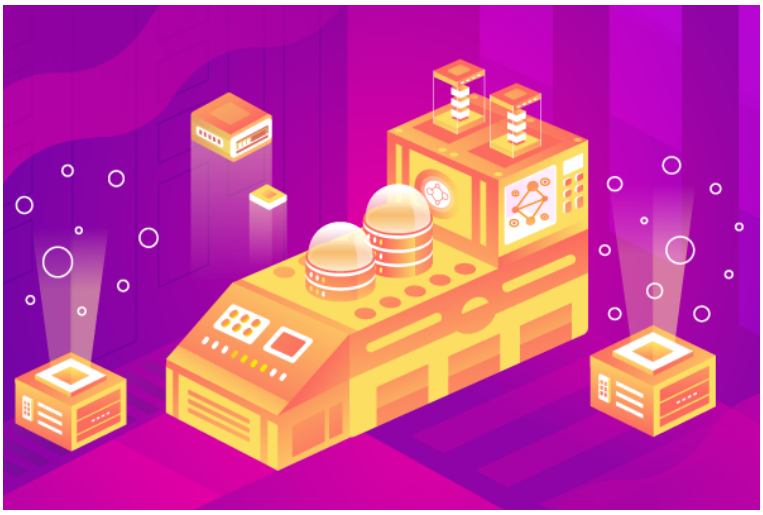
Study Guide



Certified Kubernetes Application Developer (CKAD) Study Guide

Will Boyd
February 2022

A CLOUD GURU

Contents

Application Design and Build

Building Container Images

Relevant Documentation

- Install Docker Engine on Ubuntu
- docker build
- Dockerfile reference

Exam Tips

- Images are files that include all of the software needed to run a container.
- A Dockerfile defines the contents of an image.
- The `docker build` command builds an image using a Dockerfile.

A Dockerfile is a set of instructions for building a container image with Docker.

This sample Dockerfile uses the `nginx:stable` image as a baseline. Then, it copies a file called `index.html` from the local file system into the container image at the location `/usr/share/nginx/html/`.

```
FROM nginx:stable

COPY index.html /usr/share/nginx/html/
```

- `FROM` - Sets a starting image to use as a baseline.
- `COPY` - Copies a file (`index.html`) from the local file system into the container image (in this case, at the location `/usr/share/nginx/html/`).

Build a container image from the current directory. The `-t` flag specifies the image tag name.

```
docker build -t my-image:1.0 .
```

Save an image to a file. The `-o` sets the destination file. `my-image:1.0` is the tag name of the image to save.

```
docker save -o ~/my-image.1.0.tar my-image:1.0
```

Running Jobs and CronJobs

Relevant Documentation

- Jobs
- CronJob

Exam Tips

- A Job is designed to run a containerized task successfully to completion.
- CronJobs run Jobs periodically according to a schedule.
- The `restartPolicy` for a Job or CronJob Pod must be `OnFailure` or `Never`.
- Use `activeDeadlineSeconds` in the Job spec to terminate the Job if it runs too long.

A Job executes a containerized task and attempts to run it successfully to completion.

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-job
spec:
  template:
    spec:
      containers:
      - name: print
        image: busybox:stable
        command: ["echo",  "This is a test!"]
      restartPolicy: Never
  backoffLimit: 4
  activeDeadlineSeconds: 10
```

`activeDeadlineSeconds` sets a time limit for the Job to execute. If the Job exceeds this time limit, it will be terminated.

A CronJob executes jobs regularly according to a schedule.

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-cronjob
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: print
            image: busybox:stable
            command: ["echo",  "This is a test!"]
          restartPolicy: Never
      backoffLimit: 4
      activeDeadlineSeconds: 10
```

The `schedule` is a cron expression that determines the schedule the CronJob will execute on. This example will execute once every minute.

Building Multi-Container Pods

Relevant Documentation

- **The Distributed System ToolKit: Patterns for Composite Containers**
- **Pods - Resource Sharing and Communication**
- **Shared Volumes**

Exam Tips

- A sidecar container performs some task that helps the main container.
- An ambassador container proxies network traffic to and/or from the main container.
- An adapter container transforms the main container's output.

A multi-container Pod is simply a Pod that includes more than one container.

In this example, `writer` is the main container. It writes data to a file. The `sidecar` container reads this data from a shared volume and then writes it to the container log.

```
apiVersion: v1
kind: Pod
metadata:
  name: sidecar-test
spec:
  containers:
  - name: writer
    image: busybox:stable
    command: ['sh', '-c', 'echo "The writer wrote this!" > /
output/data.txt; while true; do sleep 5; done']
    volumeMounts:
    - name: shared
      mountPath: /output
  - name: sidecar
    image: busybox:stable
    command: ['sh', '-c', 'while true; do cat /input/data.txt;
sleep 5; done']
    volumeMounts:
    - name: shared
      mountPath: /input
  volumes:
  - name: shared
    emptyDir: {}
```

Using Init Containers

Relevant Documentation

- **Init Containers**

Exam Tips

- Init containers run to completion before the main container starts up.
- Add init containers using the `initContainers` field of the PodSpec.

Init containers execute tasks during the startup phase of a Pod, before the main container begins to start up.

```
apiVersion: v1
kind: Pod
metadata:
  name: init-test
spec:
  containers:
  - name: nginx
    image: nginx:stable
  initContainers:
  - name: busybox
    image: busybox:stable
    command: ['sh', '-c', 'sleep 60']
```

Exploring Volumes

Relevant Documentation

- **Volumes**

Exam Tips

- The `volumes` field in the Pod spec defines details about volumes used in the Pod.
- The `volumeMounts` field in the container spec mounts a volume to a specific container at a specific location.
- `hostPath` volumes mount data from a specific location on the host (k8s node).
- `hostPath` volume types:
  - `Directory` – Mounts an existing directory on the host.
  - `DirectoryOrCreate` – Mounts a directory on the host, and creates it if it doesn't exist.
  - `File` – Mounts an existing single file on the host.
  - `FileOrCreate` – Mounts a file on the host, and creates it if it doesn't exist.
- `emptyDir` volumes provide temporary storage that uses the host file system and are removed if the Pod is deleted.

Volumes are used to provide external storage to containers.

This example Pod uses a hostPath volume to read data from the host machine at `/etc/hostPath`.

```
apiVersion: v1
kind: Pod
metadata:
  name: hostpath-volume-test
spec:
  restartPolicy: OnFailure
  containers:
  - name: busybox
    image: busybox:stable
    command: ['sh', '-c', 'cat /data/data.txt']
    volumeMounts:
    - name: host-data
      mountPath: /data
  volumes:
  - name: host-data
    hostPath:
      path: /etc/hostPath
      type: Directory
```

Using PersistentVolumes

Relevant Documentation

- **Persistent Volumes**

Exam Tips

- A PersistentVolume defines a storage resource.
- A PersistentVolumeClaim defines a request to consume a storage resource.
- PersistentVolumeClaims automatically bind to a PersistentVolume that meets their criteria.
- Mount a PersistentVolumeClaim to a container like a regular volume.

A PersistentVolume, like a regular volume, provides external storage to containers. However, a PersistentVolume lets you treat storage like an abstract resource, much like how Kubernetes treats other resources like CPU and memory.

A PersistentVolume:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: hostpath-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
  - ReadWriteOnce
  storageClassName: slow
  hostPath:
    path: /etc/hostPath
    type: Directory
```

In order to use a PersistentVolume, you need a PersistentVolumeClaim. The PersistentVolumeClaim will automatically bind to a PersistentVolume that meets its requirements.

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: hostpath-pvc
spec:
  accessModes:
  - ReadWriteOnce
  resources:
    requests:
      storage: 200Mi
  storageClassName: slow
```

You can mount a PersistentVolumeClaim within a Pod just like any regular volume.

```
apiVersion: v1
kind: Pod
metadata:
  name: pv-pod-test
spec:
  restartPolicy: OnFailure
  containers:
  - name: busybox
    image: busybox:stable
    command: ['sh', '-c', 'cat /data/data.txt']
    volumeMounts:
    - name: pv-host-data
      mountPath: /data
  volumes:
  - name: pv-host-data
    persistentVolumeClaim:
      claimName: hostpath-pvc
```

Application Deployment

Understanding Deployments

Relevant Documentation

- Deployment

Exam Tips

- A Deployment actively manages a desired state for a set of replica Pods.
- The Pod template provides the Pod configuration that the Deployment will use to create new Pods.
- The `replicas` field sets the number of replicas. You can scale up or down by changing this value.

A Deployment manages a desired state for a set of replica Pods and helps with the process of rolling out new code.

This example creates 2 replicas of a Pod running `nginx`.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Performing Rolling Updates

Relevant Documentation

- Updating a Deployment

Exam Tips

- A rolling update gradually rolls out changes to a Deployment's Pod template by gradually replacing replicas with new ones.
- Use `kubectl rollout status` to check the status of a rolling update.
- Roll back the latest rolling update with: `kubectl rollout undo`.

Deployments can perform rolling updates by gradually replacing old Pods with new ones running new code/configuration.

You can initiate a rolling update simply by changing the Pod template configuration (such as the image version). One way to do this is simply by editing the Deployment, for example, with `kubectl edit deployment`.

You can also use a declarative command like:

```
kubectl set image deployment.v1.apps/rolling-deployment
nginx=nginx:1.16.1
```

Check the status of the current rollout:

```
kubectl rollout status deployment/rolling-deployment
```

Undo/roll back the last rolling update:

```
kubectl rollout undo deployment/rolling-deployment
```

Deploying with Blue/Green and Canary Strategies

Relevant Documentation

- Deployment
- Service

Exam Tips

- You can use multiple Deployments to set up blue/green environments in Kubernetes.

- Use labels and selectors on Services to direct user traffic to different Pods.
- A simple way to set up a canary environment in Kubernetes is to use a Service that selects Pods from 2 different Deployments. Vary the number of replicas to direct fewer users to the canary environment.

Deployment strategies like blue/green and canary can help you improve stability while deploying new code.

- **Blue/Green** - Create a second, identical environment running the new code, test it, then point user traffic to the new environment.
- **Canary** - Create a second, identical environment running the new code, and direct a small percentage of user traffic to the new environment to verify it is working before deploying the new code for all users.

You can use Kubernetes objects like Deployments and Services to implement these deployment strategies.

A sample blue/green setup:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: blue-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: bluegreen-test
      color: blue
  template:
    metadata:
      labels:
        app: bluegreen-test
        color: blue
    spec:
      containers:
      - name: nginx
        image: linuxacademycontent/ckad-nginx:blue
        ports:
        - containerPort: 80
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: green-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: bluegreen-test
      color: green
  template:
    metadata:
      labels:
        app: bluegreen-test
        color: green
    spec:
      containers:
      - name: nginx
        image: linuxacademycontent/ckad-nginx:green
        ports:
        - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: bluegreen-test-svc
spec:
  selector:
    app: bluegreen-test
    color: blue
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

A sample canary setup:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: main-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: canary-test
      environment: main
  template:
    metadata:
      labels:
        app: canary-test
        environment: main
    spec:
      containers:
      - name: nginx
        image: linuxacademycontent/ckad-nginx:1.0.0
        ports:
        - containerPort: 80
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: canary-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: canary-test
      environment: canary
  template:
    metadata:
      labels:
        app: canary-test
        environment: canary
    spec:
      containers:
      - name: nginx
        image: linuxacademycontent/ckad-nginx:canary
        ports:
        - containerPort: 80
```

```
apiVersion: v1
kind: Service
metadata:
  name: canary-test-svc
spec:
  selector:
    app: canary-test
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
```

Installing Helm

Relevant Documentation

- **Installing Helm**

Exam Tips

- Helm is a package management tool for Kubernetes applications.

Note: Installation of the Helm tool is not part of the CKAD curriculum.

Using Helm

Relevant Documentation

- **Helm Quickstart Guide**

Exam Tips

- Helm Charts are packages that contain all of the resource definitions needed to get an application up and running in a cluster.
- A Helm Repository is a collection of Charts and a source for browsing and downloading them.

Before installing a Helm Chart, you need to add a repository.

```
helm repo add bitnami https://charts.bitnami.com/bitnami
```

Update a repository.

```
helm repo update
```

View a list of charts available in a repository.

```
helm search repo bitnami
```

Install a chart.

```
helm install --set persistence.enabled=false -n dokuwiki
dokuwiki bitnami/docuwiki
```

Application Observability and Maintenance

Understanding the API Deprecation Policy

Relevant Documentation

- Kubernetes Deprecation Policy
- Deprecated API Migration Guide

Exam Tips

- API Deprecation is the process of announcing changes to an API early, giving users time to update their code and/or tools.
- Kubernetes removes support for deprecated APIs that are in GA (General Availability) only after 12 months or 3 Kubernetes releases, whichever is longer.

Implementing Probes and Health Checks

Relevant Documentation

- Configure Liveness, Readiness, and Startup Probes

Exam Tips

- Liveness probes check if a container is healthy so that it can be restarted if it is not.
- Readiness probes check whether a container is fully started up and ready to be used.
- Probes can run a command inside the container, make an HTTP request, or attempt a TCP socket connection to determine container status.

A liveness probe that runs a command to detect container health status:

```
apiVersion: v1
kind: Pod
metadata:
  name: liveness-pod
spec:
  containers:
  - name: busybox
    image: busybox:stable
    command: ['sh', '-c', 'while true; do sleep 10; done']
    livenessProbe:
      exec:
        command: ['echo', 'health check!']
      initialDelaySeconds: 5
      periodSeconds: 5
```

This Pod has both a liveness probe and a readiness probe, both of which use an `http` request to check the status of the container.

```
apiVersion: v1
kind: Pod
metadata:
  name: readiness-pod
spec:
  containers:
  - name: nginx
    image: nginx:1.20.1
    ports:
    - containerPort: 80
    livenessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 3
      periodSeconds: 3
    readinessProbe:
      httpGet:
        path: /
        port: 80
      initialDelaySeconds: 15
      periodSeconds: 5
```

Monitoring Kubernetes Applications

Relevant Documentation

- Resource Metrics Pipeline
- Tools for Monitoring Resources
- metrics-server GitHub

Exam Tips

- The Kubernetes metrics API provides metric data about container performance.
- You can view Pod metrics using `kubectl top pod`.

When metrics server is installed, you can use `kubectl top` to view resource usage data.

View resource usage for Pods in the `default` Namespace:

```
kubectl top pod
```

You can use `-n` to specify the Namespace:

```
kubectl top pos -n one
```

View resource usage by node:

```
kubectl top node
```

Accessing Container Logs

Relevant Documentation

- Logging Architecture

Exam Tips

- Standard/error output for containers is stored in the container log.
- You can view the container log using `kubectl logs`.
- For multi-container Pods, use the `-c` flag to specify which container's logs you want to view.

Use `kubectl logs` to get container logs.

```
kubectl logs my-pod -n one
```

Use `-c` to specify which container to get logs for. This is required if the Pod has more than one container.

```
kubectl logs my-pod -n one -c busybox
```

Debugging in Kubernetes

Relevant Documentation

- **Troubleshoot Applications**
- **Application Introspection and Debugging**
- **Monitoring, Logging, and Debugging**

Exam Tips

- Use `kubectl get pods` to check the status of all Pods in a Namespace. Use the `--all-namespaces` flag if you don't know what Namespace to look in.
- Use `kubectl describe` to get detailed information about Kubernetes objects.
- Use `kubectl logs` to retrieve container logs.
- Check cluster-level logs if you still cannot locate any relevant information.

List Pods to view Pod status within a Namespace:

```
kubectl get pods
```

Use `--all-namespaces` to check Pod status in all Namespaces. This is useful if you do not know which Namespace a problem is located in.

```
kubectl get pods --all-namespaces
```

Get detailed information about a single Pod:

```
kubectl describe pod my-pod
```

Get container logs to diagnose issues:

```
kubectl logs my-pod
```

Get Kubernetes API Server logs for kubeadm cluster (must be run on a control plane node):

```
sudo cat /var/log/containers/kube-apiserver-k8s-control_kube-system_kube-apiserver-<hash>.log
```

Get kubelet logs for the current node:

```
sudo journalctl -u kubelet
```

Application Environment, Configuration, and Security

Using Custom Resources (CRD)

Relevant Documentation

- Extend the Kubernetes API with CustomResourceDefinitions
- Custom Resources

Exam Tips

- Custom resources are extensions of the Kubernetes API.
- A CustomResourceDefinition defines a custom resource.

An example of a CRD:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: beehives.acloud.guru
spec:
  group: acloud.guru
  names:
    plural: beehives
    singular: beehive
    kind: BeeHive
    shortNames:
    - hive
  scope: Namespaced
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
```

```
              supers:
                type: integer
              bees:
                type: integer
```

The example CRD would allow you to create objects like this:

```
apiVersion: acloud.guru/v1
kind: BeeHive
metadata:
  name: test-beehive
spec:
  supers: 3
  bees: 60000
```

You can interact with custom resources, just like regular Kubernetes resources, with commands like `kubectl get` and `kubectl describe`.

Using ServiceAccounts

Relevant Documentation

- **Configure Service Accounts for Pods**
- **Managing Service Accounts**
- **Using RBAC Authorization**

Exam Tips

- ServiceAccounts allow processes within containers to authenticate with the Kubernetes API Server.
- You can set the Pod's ServiceAccount with `serviceAccountName` in the Pod spec.
- The Pod's ServiceAccount token is automatically mounted to the Pod's containers.

An example ServiceAccount:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-sa
automountServiceAccountToken: true
```

Set the ServiceAccount used by a Pod with `spec.serviceAccountName`.

```
apiVersion: v1
kind: Pod
metadata:
  name: sa-pod
spec:
  serviceAccountName: my-sa
  containers:
  - name: nginx
    image: nginx:stable
```

Understanding Kubernetes Auth

Relevant Documentation

- **Authenticating**
- **Controlling Access to the Kubernetes API**
- **Using RBAC Authorization**

Exam Tips

- Normal users usually authenticate using client certificates, while ServiceAccounts usually use tokens.
- Authorization for both normal users and ServiceAccounts can be managed using Role-Based Access Control (RBAC).
- Roles and ClusterRoles define a specific set of permissions.
- RoleBindings and ClusterRoleBindings tie Roles or ClusterRoles to users/ServiceAccounts.

Role-Based Access Control (RBAC) provides authorization for the Kubernetes API, determining the permissions allowed to users, ServiceAccounts, etc.

- `Role` - Defines a set of permissions, and exists within a Namespace.
- `ClusterRole` - Defines a set of permissions, and exists cluster-wide.
- `RoleBinding` - Binds a `Role` or `ClusterRole` to subjects such as users or ServiceAccounts. The permissions take effect only within a Namespace.
- `ClusterRoleBinding` - Binds a `Role` or `ClusterRole` to subjects such as users or ServiceAccounts. The permission take effect cluster-wide.

A role that provides permission to get a list of Pods:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: list-pods-role
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["list"]
```

A RoleBinding that binds the Role `list-pods-role` to the ServiceAccount `my-sa` in the `default` Namespace.

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: list-pods-rb
subjects:
- kind: ServiceAccount
  name: my-sa
  namespace: default
roleRef:
  kind: Role
  name: list-pods-role
  apiGroup: rbac.authorization.k8s.io
```

Exploring Admission Control

Relevant Documentation

- **Using Admission Controllers**
- **A Guide to Kubernetes Admission Controllers**

Exam Tips

- Admission controllers intercept requests to the Kubernetes API and can be used to validate and/or modify them.
- You can enable admission controllers using the `--enable-admission-plugins` flag for kube-apiserver.

Admission Controllers act upon incoming requests to the Kubernetes API. They can validate/deny and even modify requests.

```
sudo vi /etc/kubernetes/manifests/kube-apiserver.yaml
```

The `--enable-admission-plugins` flag for Kube API Server allows you to enable additional admission plugins. The `NamespaceAutoProvision` admission controller plugin automatically creates Namespaces when you try to create an object with a Namespace that does not exist.

```
    - --enable-admission-
plugins=NodeRestriction,NamespaceAutoProvision
```

Managing Compute Resource Usage

Relevant Documentation

- **Managing Resources for Containers**
- **Assign CPU Resources to Containers and Pods**
- **Resource Quotas**

Exam Tips

- A resource request informs the cluster of the expected resource usage for a container. It is used to select a node that has enough resources available to run the Pod.
- A resource limit sets an upper limit on how many resources a container can use. If the container process attempts to go above this limit, the container process will be terminated.
- A ResourceQuota limits the amount of resources that can be used within a specific Namespace. If a user attempts to create or modify objects in that Namespace such that the quota would be exceeded, the request will be denied.

Resource Requests - Specify an approximate amount of expected resource usage. Kubernetes will use this information to schedule Pods on Nodes where the requested resources are available.

Resource Limits - Specify an enforced upper limit for resource usage. The container process will be terminated if it exceeds these limits.

A Pod with resource requests and limits for `cpu` and `memory` .

```
apiVersion: v1
kind: Pod
metadata:
  name: resources-pod
  namespace: resources-test
spec:
  containers:
  - name: busybox
    image: busybox:stable
    command: ['sh', '-c', 'while true; do echo Running...;
sleep 5; done']
    resources:
      requests:
        memory: 64Mi
        cpu: 250m
      limits:
        memory: 128Mi
        cpu: 500m
```

ResourceQuota - Specify resource usage limits for a Namespace. When creating Pods, the ResourceQuota will check the Pod's resource requests alongside any existing Pods. It will deny Pod creation if it would cause the Namespace quota to be exceeded.

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: resources-test-quota
  namespace: resources-test
spec:
  hard:
    requests.memory: 128Mi
    requests.cpu: 500m
    limits.memory: 256Mi
    limits.cpu: "1"
```

Configuring Applications with ConfigMaps and Secrets

Relevant Documentation

- ConfigMaps
- Secrets

Exam Tips

- A ConfigMap stores configuration data that can be passed to containers.
- A Secret is designed to store sensitive configuration data such as passwords or API keys.
- Data from both ConfigMaps and Secrets can be passed to containers using either a volume mount or environment variables.

ConfigMaps store configuration data to be passed to application containers.

An example ConfigMap:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
data:
  message: Hello, World!
  app.cfg: |
    # A configuration file!
    key1=value1
    key2=value2
```

You can pass ConfigMap data to a container either as an environment variable or as a mounted volume:

```
apiVersion: v1
kind: Pod
metadata:
  name: cm-pod
spec:
  restartPolicy: Never
  containers:
  - name: busybox
    image: busybox:stable
    command: ['sh', '-c', 'echo $MESSAGE; cat /config/app.cfg']
    env:
    - name: MESSAGE
      valueFrom:
        configMapKeyRef:
          name: my-configmap
          key: message
    volumeMounts:
    - name: config
      mountPath: /config
      readOnly: true
  volumes:
  - name: config
    configMap:
      name: my-configmap
      items:
      - key: app.cfg
        path: app.cfg
```

Secrets are similar to ConfigMaps, but are designed to store sensitive data like passwords or API keys.

When creating a Secret manifest, you must first base64-encode any secret values.

```
echo Secret Stuff! | base64
```

An example Secret with base64-encoded values:

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  sensitive.data: U2VjcmV0IFN0dWZmIQo=
  passwords.txt: U2VjcmV0IHN0dWZmIGluIGEgZmlsZSEK
```

You can pass Secret data to a container using environment variables or mounted volumes, just like a ConfigMap.

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-pod
spec:
  restartPolicy: Never
  containers:
  - name: busybox
    image: busybox:stable
    command: ['sh', '-c', 'echo $SENSITIVE_STUFF; cat /config/
passwords.txt']
    env:
    - name: SENSITIVE_STUFF
      valueFrom:
        secretKeyRef:
          name: my-secret
          key: sensitive.data
    volumeMounts:
    - name: secret-config
      mountPath: /config
      readOnly: true
  volumes:
  - name: secret-config
    secret:
      secretName: my-secret
      items:
      - key: passwords.txt
        path: passwords.txt
```

Configuring SecurityContext for Containers

Relevant Documentation

- [Configure a Security Context for a Pod or Container]

Exam Tips

- A container's SecurityContext allows you to control advanced security-related settings for the container.
- Set the container's user ID (UID) and group ID (GID) with `securitContext.runAsUser` and `securityContext.runAsGroup`.
- Enable or disable privilege escalation with `securityContext.allowPrivilegeEscalation`.
- Make the container root filesystem read-only with `securityContext.readOnlyRootFilesystem`.

`securityContext` allows you to customize a variety of OS-level, security-related settings for containers.

A Pod that uses `securityContext`:

```
apiVersion: v1
kind: Pod
metadata:
  name: securitycontext-pod
spec:
  containers:
  - name: busybox
    image: busybox:stable
    command: ['sh', '-c', 'while true; do echo Running...;
sleep 5; done']
    securityContext:
      runAsUser: 3000
      runAsGroup: 4000
      allowPrivilegeEscalation: false
      readOnlyRootFilesystem: true
```

- `runAsUser: 3000` - The container process will run as user ID `3000`.
- `runAsGroup: 4000` - The container process will run as group ID `4000`.
- `allowPrivilegeEscalation: false` - Disables privileged mode for the container.
- `readOnlyRootFilesystem: true` - Marks the container's root filesystem read-only, preventing any writes to that filesystem.

Services and Networking

Controlling Network Access with NetworkPolicies

Relevant Documentation

- NetworkPolicies
- Cluster Networking

Exam Tips

- If a Pod is not selected by any NetworkPolicies, the Pod is non-isolated, and all traffic is allowed.
- If a Pod is selected by any NetworkPolicy, traffic will be blocked unless it is allowed by at least 1 NetworkPolicy that selects the Pod.
- If you combine a `namespaceSelector` and a `podSelector` within the same rule, the traffic must meet both the Pod- and Namespace-related conditions in order to be allowed.
- Even if a NetworkPolicy allows outgoing traffic from the source Pod, NetworkPolicies could still block the same traffic when it is incoming to the destination Pod.

NetworkPolicies allow you to control what traffic is and is not allowed within the cluster network.

Important things to remember:

- If no NetworkPolicies select a Pod, the Pod is **non-isolated**. It allows traffic to and from itself.
- If a Pod is selected by at least 1 NetworkPolicy, it is isolated. In order for traffic to be allowed, at least 1 NetworkPolicy that selects the Pod must allow the traffic.

A default deny NetworkPolicy disables all traffic by default, leaving it up to other NetworkPolicies to specifically allow desired traffic. Note that the empty podSelector `{}` selects all Pods in the Namespace.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-test-a-default-deny-ingress
  namespace: np-test-a
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

This policy affects only `Ingress` (incoming) traffic. It allows traffic from any Pod that meets **both** of the following criteria:

- In a Namespace with the `team=bteam` label.
- The Pod has the label `app: np-test-client`.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-test-client-allow
  namespace: np-test-a
spec:
  podSelector:
    matchLabels:
      app: np-test-server
  policyTypes:
  - Ingress
  ingress:
  - from:
    - namespaceSelector:
        matchLabels:
          team: bteam
      podSelector:
        matchLabels:
          app: np-test-client
    ports:
    - protocol: TCP
      port: 80
```

This policy applies to `Egress` (outgoing) traffic. It allows traffic to Pods in any Namespace with the label `team=ateam`.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: np-test-client-allow-egress
  namespace: np-test-b
spec:
  podSelector:
    matchLabels:
      app: np-test-client
  policyTypes:
  - Egress
  egress:
  - to:
    - namespaceSelector:
        matchLabels:
          team: ateam
    ports:
    - protocol: TCP
      port: 80
```

Exploring Services

Relevant Documentation

- **Service**

Exam Tips

- Services allow you to expose an application running in multiple Pods.
- ClusterIP Services expose the Pods to other applications within the cluster.
- NodePort Services expose the Pods externally using a port that listens on every node in the cluster.

Services expose applications, directing traffic to backend Pods.

A `ClusterIP` Service is focused on exposing to other Pods in the cluster. It provides an IP address and hostnames within the cluster network that other Pods can use to access the Service.

```
apiVersion: v1
kind: Service
metadata:
  name: clusterip-service
spec:
  type: ClusterIP
  selector:
    app: service-server
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
```

A `NodePort` Service can expose an application externally. It listens on a port on each node in the cluster.

```
apiVersion: v1
kind: Service
metadata:
  name: nodeport-service
spec:
  type: NodePort
  selector:
    app: service-server
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 80
      nodePort: 30080
```

Exposing Applications with Ingress

Relevant Documentation

- **Ingress**

Exam Tips

- An Ingress manages external access to Kubernetes applications.
- An Ingress routes to 1 or more Kubernetes Services.

- You need an Ingress controller to implement the Ingress functionality. Which controller you use determines the specifics of how the Ingress will work.

An **Ingress** is designed to expose applications externally. Usually, an Ingress routes traffic to a Service backend. An Ingress can also provide additional features such as SSL termination.

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-test-ingress
spec:
  ingressClassName: nginx
  rules:
  - host: ingresstest.acloud.guru
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: ingress-test-service
            port:
              number: 80
```