Kubernetes Essential Yaml files

Introduction to Kubernetes YAML Files

Kubernetes YAML files are essential for defining resources and configurations for applications running in a Kubernetes cluster. These files help describe how various resources should behave and interact within the cluster.

1. **Pod Definition (pod.yaml)** – Defines individual pods with your containers.
2. **Deployment (deployment.yaml)** – Manages deployments, ensuring the right number of pods are running.
3. **Service (service.yaml)** – Exposes your pods and allows communication between them.
4. **ConfigMap (configmap.yaml)** – Stores configuration data.
5. **Secret (secret.yaml)** – Stores sensitive data like passwords, tokens, etc.
6. **Namespace (namespace.yaml)** – Provides isolation for different parts of your system.
7. **PersistentVolume (pv.yaml)** – Represents storage resources.
8. **PersistentVolumeClaim (pvc.yaml)** – Requests storage resources from PVs.
9. **Ingress (ingress.yaml)** – Manages HTTP/HTTPS routing and external access to your services.

10. **HorizontalPodAutoscaler (hpa.yaml)** – Automatically adjusts the number of pods based on resource usage.

11. **StatefulSet (statefulset.yaml)** – Manages stateful applications where pods need stable, unique identities.

12. **DaemonSet (daemonset.yaml)** – Ensures one pod runs on each node in the cluster.

13. **Job (job.yaml)** – Executes batch processing tasks.

14. **CronJob (cronjob.yaml)** – Schedules periodic tasks, like cron jobs in Unix/Linux.

15. **NetworkPolicy (networkpolicy.yaml)** – Controls network access between pods and services.

16. **ResourceQuota (resourcequota.yaml)** – Limits the resources a namespace can consume.

17. **PodDisruptionBudget (pdb.yaml)** – Ensures a certain number of pods are always available during voluntary disruptions (like draining nodes for maintenance).

18. **Role and RoleBinding (for RBAC)**
19. **NetworkPolicy**
20. **ServiceAccount**

Common Kubernetes YAML Files and Their Purpose

1. Pod Definition (pod.yaml)

A **Pod** is the smallest deployable unit in Kubernetes, representing a single instance of a running process in a cluster. A Pod can contain one or more containers that share the same storage, network, and specification. Pods allow containers to communicate with each other easily and share resources such as volumes.

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
    - name: my-container
      image: nginx
```

- **apiVersion**: Specifies the version of the Kubernetes API.
- **kind**: Defines the type of resource (here, it is a Pod).
- **metadata**: Contains the name and other details of the Pod.
- **spec**: Defines the desired state of the Pod (container running the nginx image).

---

2. Deployment (deployment.yaml) – Manages deployments, ensuring the right number of pods are running.

A **Deployment** is a higher-level abstraction that manages **Pods**. It ensures the correct number of Pods are running and provides updates or rollback mechanisms. Deployments are great for stateless applications where you want to manage scaling and versioning easily.

yaml

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

- **replicas**: Specifies the number of Pod replicas to run.
- **selector**: Matches Pods with the label app: my-app.
- **template**: Defines the Pod template for creating Pods.
- **containers**: Defines the container(s) inside the Pod.

---

3. Service (service.yaml) – Exposes your pods and allows communication between them.

A **Service** exposes a set of Pods to enable communication between them. It acts as a load balancer that distributes traffic across the Pods, providing a stable endpoint (DNS or IP) even when Pods are added or removed. Services can expose Pods internally (within the cluster) or externally (to the outside world)..

yaml

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

- **selector**: Specifies which Pods the Service targets by matching labels.
- **ports**: Defines the ports for the service (here, exposing port 80).
- **type**: Defines the Service type (ClusterIP makes the service accessible only within the cluster).

---

4. ConfigMap (configmap.yaml) – Stores configuration data.

A **ConfigMap** is used to store non-sensitive configuration data in key-value pairs. It allows your application to access configuration data without hardcoding it into the codebase. ConfigMaps are often used to store configuration values, environment variables, or command-line arguments.

yaml

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: my-configmap
```

```
data:
  key1: value1
  key2: value2
```

- **data**: Stores configuration data as key-value pairs.

---

5. Secret (secret.yaml) – Stores sensitive data like passwords, tokens, etc.

A **Secret** is used to store sensitive data, like passwords, API keys, or tokens. Secrets are stored in a more secure way compared to ConfigMaps, and they can be mounted into Pods or passed as environment variables securely. They can be encrypted in etcd to prevent unauthorized access.

yaml

```
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
type: Opaque
data:
  username: dXNlcm5hbWU=  # base64 encoded "username"
  password: cGFzc3dvcmQ=  # base64 encoded "password"
```

- **data**: Contains base64-encoded sensitive data.

---

6. Namespace (namespace.yaml) – Provides isolation for different parts of your system.

A **Namespace** provides a way to isolate resources in a Kubernetes cluster. It's like creating separate virtual clusters within the same physical cluster. Namespaces are used to group resources logically, enabling multiple teams or applications to coexist in the same cluster without interfering with each other.

yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-namespace
```

- **name**: Defines the name of the Namespace.

7. PersistentVolume (pv.yaml) – Represents storage resources.

A **PersistentVolume** (PV) represents a piece of storage in the cluster, either from a network-attached storage (NAS) device, a cloud disk, or local storage. PVs are managed by Kubernetes and can be dynamically provisioned or manually created.

yaml

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: /mnt/data
```

- **capacity**: Defines the storage size.
- **accessModes**: Specifies how the volume can be accessed.
- **hostPath**: Specifies the path on the host node where the volume is located.

---

8. PersistentVolumeClaim (pvc.yaml) – Requests storage resources from PVs.

A **PersistentVolumeClaim** (PVC) is a request for storage by a user. It specifies the desired storage size and access modes. When a PVC is created, Kubernetes binds it to a suitable PV that satisfies the request, ensuring that storage is available for Pods.

yaml

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
```

- **resources**: Defines the requested storage size.

---

9. Ingress (ingress.yaml) – Manages HTTP/HTTPS routing and external access to your services.

An **Ingress** is used to manage external access to services within the cluster, typically HTTP or HTTPS traffic. It allows for routing based on URL paths or domain names. Ingress controllers handle traffic routing and can be configured with SSL termination and load balancing.

yaml

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

- **rules**: Defines the routing rules. Traffic to myapp.example.com is routed to the my-service service on port 80.

---

10. HorizontalPodAutoscaler (hpa.yaml) – Automatically adjusts the number of pods based on resource usage.

The **HorizontalPodAutoscaler** (HPA) automatically adjusts the number of Pods in a Deployment or StatefulSet based on observed resource usage (CPU, memory,

etc.). It ensures that the application scales up or down to match the demand, providing efficient resource usage.

yaml

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: my-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: my-deployment
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: cpu
        target:
          type: Utilization
          averageUtilization: 50
```

- **scaleTargetRef**: Specifies the resource (e.g., Deployment) to scale.
- **metrics**: Defines the scaling criteria, such as CPU utilization.

Advanced Kubernetes Resources

11. StatefulSet (statefulset.yaml) – Manages stateful applications where pods need stable, unique identities.

A **StatefulSet** is used for managing stateful applications where Pods require persistent storage and stable, unique network identities. Unlike Deployments, StatefulSets ensure that each Pod gets a unique identifier and persistent storage, which is important for databases or other applications with persistent state.

yaml

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-statefulset
spec:
  serviceName: "my-statefulset"
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-container
          image: nginx
          volumeMounts:
            - name: my-volume
              mountPath: /data
  volumeClaimTemplates:
    - metadata:
        name: my-volume
      spec:
```

```
    accessModes: [ "ReadWriteOnce" ]
    resources:
      requests:
        storage: 1Gi
```

- **volumeClaimTemplates**: Defines persistent storage for each replica in the StatefulSet.

---

12. DaemonSet (daemonset.yaml) – Ensures one pod runs on each node in the cluster.

A **DaemonSet** ensures that a copy of a specific Pod runs on all (or selected) nodes in a Kubernetes cluster. It's useful for background tasks like logging, monitoring, or networking that need to run on every node.

yaml

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: my-daemonset
spec:
  selector:
    matchLabels:
      name: my-daemonset
  template:
    metadata:
      labels:
        name: my-daemonset
    spec:
```

```
  containers:
    - name: my-container
      image: nginx
```

- **selector**: Ensures that the DaemonSet applies to the correct Pods.

---

13. Job (job.yaml) – Executes batch processing tasks.

A **Job** runs a single or batch processing task to completion. It creates one or more Pods and ensures that the task is completed successfully. Jobs are useful for tasks that run until they finish, like batch processing or data migration.

yaml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: my-job
spec:
  template:
    spec:
      containers:
        - name: my-container
          image: busybox
          command: ["echo", "Hello, Kubernetes!"]
      restartPolicy: Never
```

- **restartPolicy**: Ensures the Job runs to completion without restarting the container.

---

14. CronJob (cronjob.yaml) – Schedules periodic tasks, like cron jobs in Unix/Linux.

A **CronJob** runs jobs on a scheduled basis, similar to cron jobs in Linux. You can specify a schedule (using cron syntax) to run tasks periodically, such as backups or reports.

yaml

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: my-cronjob
spec:
  schedule: "*/5 * * * *"  # Runs every 5 minutes
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: my-container
              image: busybox
              command: ["echo", "Hello, CronJob!"]
          restartPolicy: OnFailure
```

- **schedule**: Defines the cron expression for scheduling the job.

---

15. NetworkPolicy (networkpolicy.yaml) – Controls network access between pods and services.

A **NetworkPolicy** controls the traffic flow between Pods in a Kubernetes cluster. It allows you to define rules that specify which Pods can communicate with each other, providing network-level security.

yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: my-network-policy
spec:
  podSelector:
    matchLabels:
      app: my-app
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: frontend
  egress:
    - to:
        - podSelector:
            matchLabels:
              role: backend
```

- **ingress**: Defines allowed incoming traffic from Pods with specific labels.
- **egress**: Defines allowed outgoing traffic to Pods with specific labels.

---

16. ResourceQuota (resourcequota.yaml) – Limits the resources a namespace can consume.

A **ResourceQuota** limits the amount of resources (like CPU, memory, or storage) that can be consumed by resources within a specific namespace. It helps to prevent any one namespace from consuming too many resources, ensuring fair distribution within the cluster.

yaml

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: my-resource-quota
spec:
  hard:
    requests.cpu: "4"
    requests.memory: 4Gi
    limits.cpu: "8"
    limits.memory: 8Gi
```

- **hard**: Specifies the resource limits for the namespace.

---

17. PodDisruptionBudget (pdb.yaml) – Ensures a certain number of pods are always available during voluntary disruptions (like draining nodes for maintenance).

A **PodDisruptionBudget** ensures that a minimum number of Pods are always available, even during voluntary disruptions, such as node maintenance. It helps maintain application availability during planned operations like scaling down or upgrading nodes.

yaml

```
apiVersion: policy/v1
```

```
kind: PodDisruptionBudget
metadata:
  name: my-pdb
spec:
  minAvailable: 2
  selector:
    matchLabels:
      app: my-app
```

- **minAvailable**: Ensures that at least two Pods remain available during disruptions.

---

18. Role and RoleBinding (for RBAC)

In Kubernetes, **Role-Based Access Control (RBAC)** is used to manage who can do what within your cluster. You control permissions (like which actions a user or service can perform on which resources) through **Roles** and **RoleBindings**.

- **Role**: Defines the permissions (like creating or deleting pods) that apply within a **namespace**.

- **RoleBinding**: Links a **Role** to a specific user or service account, giving them the permissions defined in the Role.

Example:

Let's say you want to give someone permission to view (get or list) the pods in a particular namespace. You would define a **Role** that grants these permissions, and then use a **RoleBinding** to assign that Role to a user.

role.yaml

yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: default  # The namespace where the role applies
  name: pod-reader   # The name of the role
rules:
  - apiGroups: [""]
    resources: ["pods"]
    verbs: ["get", "list"]  # Permissions to view (get or list) pods
```

rolebinding.yaml

yaml

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: read-pods
  namespace: default
subjects:
  - kind: User
    name: "john"  # The user who gets the permissions
    apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader  # The role we created earlier
  apiGroup: rbac.authorization.k8s.io
```

With this, user john is granted the permission to **read** the pods in the default namespace.

---

19. NetworkPolicy

A **NetworkPolicy** controls how pods can communicate with each other and other services in the Kubernetes cluster. This is useful when you want to limit which pods can access others, providing security.

Example:

Let's say you have two sets of pods: one running a web app (frontend) and the other running a backend API. You might only want the frontend pods to communicate with the backend pods. A **NetworkPolicy** can be used to enforce this rule.

networkpolicy.yaml

yaml

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-internal
  namespace: default
spec:
  podSelector:
    matchLabels:
      app: myapp  # This applies to pods with the label app=myapp
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: frontend  # Only allow connections from frontend pods
```

With this policy, pods labeled app=myapp can only receive traffic from pods labeled app=frontend. This helps secure your network by limiting connections between pods.

---

20. ServiceAccount

A **ServiceAccount** is an identity used by **pods** to interact with the Kubernetes API or other services. Each pod can be assigned a ServiceAccount, which determines its permissions based on the roles assigned to that ServiceAccount.

Example:

If you have a pod that needs to access the Kubernetes API (for example, to list pods), you would create a **ServiceAccount** and assign it to the pod.

serviceaccount.yaml

yaml

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: my-service-account  # Name of the service account
  namespace: default        # The namespace where it will be used
```

You can then assign this ServiceAccount to a pod:

pod.yaml

yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  serviceAccountName: my-service-account  # Link the service account to the pod
  containers:
  - name: my-container
```

```
    image: my-image
```

By assigning a **ServiceAccount** to the pod, you can control what the pod is allowed to do in the cluster. For example, if the ServiceAccount has a **Role** that allows reading pods, then the pod can access the pod list.
