Kubernetes scenario based questions

---

### 1. Scenario: You want to deploy an application with multiple replicas. How do you scale your Kubernetes deployment?

**Answer:** You can scale a Kubernetes deployment by using the `kubectl scale` command or by modifying the deployment's YAML file.

Using `kubectl` command:

```
kubectl scale deployment <deployment-name> --replicas=<number-of-replicas>
```

For example:

```
kubectl scale deployment my-app --replicas=5
```

Alternatively, you can edit the YAML file of the deployment and change the `replicas` field:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 5
```

Apply the changes:

```
kubectl apply -f deployment.yaml
```

---

### 2. Scenario: You have a Kubernetes pod that is stuck in `CrashLoopBackOff`. How do you diagnose and fix it?

**Answer:**

1. **Check pod logs** to understand why it’s crashing:

   ```
   kubectl logs <pod-name> --previous
   ```

2. **Describe the pod** to get detailed information about its state and events:

   ```
   kubectl describe pod <pod-name>
   ```

3. Look for any error messages or failed container states in the logs. Common issues include misconfigurations, missing files, or incorrect image versions.

4. **Fix the issue** based on the logs (e.g., fix environment variables, update the image, or modify the configuration) and then restart the pod.

To restart the pod manually:

```
kubectl delete pod <pod-name>
```

---

### 3. Scenario: You need to expose a service in your Kubernetes cluster to the internet. How do you expose the service using a LoadBalancer?

**Answer:** To expose a service using a `LoadBalancer`, you need to define a service of type `LoadBalancer` in your YAML configuration.

Example `Service` definition:

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

Apply the YAML:

```
kubectl apply -f service.yaml
```

Once the service is created, Kubernetes will provision a cloud load balancer (if supported by the cloud provider) and assign an external IP address to access the service.

---

### 4. Scenario: You want to run a job in Kubernetes that executes once and then completes. How do you create a job in Kubernetes?

**Answer:** To create a one-time job, define a `Job` resource. Here's an example of a simple Kubernetes job definition:

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

This job will run the `echo` command once and then exit.

Apply the job:

```
kubectl apply -f job.yaml
```

You can check the status of the job using:

```
kubectl get jobs
```

To view the job's logs:

```
kubectl logs job/my-job
```

---

### 5. Scenario: You need to restrict access to a Kubernetes service based on labels. How do you achieve this?

**Answer:** You can use **Network Policies** to restrict traffic based on pod labels. A network policy controls the communication between pods based on labels and namespaces.

Example of a `NetworkPolicy` that allows traffic only to pods with the label `role=db` in the same namespace:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-db
spec:
  podSelector:
    matchLabels:
      role: db
  ingress:
  - from:
    - podSelector:
        matchLabels:
          role: app
```

This policy allows only pods with the `role=app` label to communicate with pods labeled `role=db`.

Apply the network policy:

```
kubectl apply -f networkpolicy.yaml
```

---

### 6. Scenario: You want to upgrade a deployment to a new version. How do you perform a rolling update in Kubernetes?

**Answer:** Kubernetes supports **rolling updates** by default when you update a `Deployment`. To upgrade to a new version, simply update the deployment's container image.

1. Update the image version in the deployment YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
```

```
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: my-app:v2  # New version of the image
        ports:
        - containerPort: 80
```

2. Apply the update:

```
kubectl apply -f deployment.yaml
```

Kubernetes will automatically perform a rolling update by replacing the old pods with the new ones without downtime.

---

### 7. Scenario: You want to increase the memory limit of a pod. How do you modify the pod's resource limits?

**Answer:** To modify the resource limits (CPU/memory) for a pod, you need to update the resource requests and limits in the pod’s YAML definition.

Example modification:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
    resources:
      requests:
        memory: "64Mi"
        cpu: "250m"
      limits:
        memory: "128Mi"
        cpu: "500m"
```

Apply the updated YAML:

```
kubectl apply -f pod.yaml
```

To ensure the pod is restarted with the new limits, you can delete the pod or trigger a deployment update.

---

### 8. Scenario: You need to monitor the health of a pod. How do you configure a liveness and readiness probe?

**Answer:** Liveness and readiness probes are configured within the pod's container specification. These probes check the health of the application running inside the container.

Example of liveness and readiness probes:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
    livenessProbe:
      httpGet:
        path: /healthz
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
    readinessProbe:
      httpGet:
        path: /readiness
        port: 80
      initialDelaySeconds: 5
      periodSeconds: 10
```

- **Liveness Probe** checks if the container is still running. If it fails, the container will be restarted.
- **Readiness Probe** checks if the container is ready to handle traffic. If it fails, Kubernetes will stop routing traffic to that pod.

---

### 9. Scenario: You want to create a persistent volume for a pod to store data. How do you configure a persistent volume and claim in Kubernetes?

**Answer:**

1. **Create a PersistentVolume (PV)**:

   ```
   apiVersion: v1
   kind: PersistentVolume
   metadata:
     name: my-pv
   spec:
     capacity:
       storage: 1Gi
     accessModes:
       - ReadWriteOnce
     persistentVolumeReclaimPolicy: Retain
   ```

```
  hostPath:
    path: /mnt/data
```

2. **Create a PersistentVolumeClaim (PVC)** to request the storage:

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

3. **Use the PVC in a pod**:

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - name: my-container
    image: nginx
    volumeMounts:
    - mountPath: /mnt/data
      name: my-storage
  volumes:
  - name: my-storage
    persistentVolumeClaim:
      claimName: my-pvc
```

This configuration ensures that the pod has access to persistent storage.

---

### 10. Scenario: You want to limit the CPU and memory usage for a container. How do you set resource requests and limits?

**Answer:** In the pod or container spec, you can define **resource requests** (minimum resources) and **limits** (maximum resources) for CPU and memory.

Example:

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

```
    requests:
      memory: "64Mi"
      cpu: "250m"
    limits:
      memory: "128Mi"
      cpu: "500m"
```

- **Requests**: Kubernetes uses these values to schedule the pod (ensures the pod gets enough resources).
- **Limits**: Kubernetes enforces these limits (pod will be killed if it exceeds them).

---

### 11. Scenario: You have a Kubernetes cluster with multiple worker nodes. One of the nodes becomes unresponsive and needs to be replaced. Explain the steps you would take to replace the node without affecting the availability of applications running on the cluster.

**Answer:**

To replace the unresponsive node without affecting application availability, I would follow these steps:

1. Drain the unresponsive node: Use the kubectl drain command to gracefully evict all the pods running on the unresponsive node. This ensures that the pods are rescheduled on other healthy nodes.
2. Cordon the unresponsive node: Use the kubectl cordon command to mark the node as unschedulable. This prevents new pods from being scheduled on the node while it's being replaced.
3. Remove the unresponsive node: Once all the pods are safely rescheduled, you can remove the unresponsive node from the cluster, either by repairing it or provisioning a new node.
4. Uncordon the node: Once the new node is ready, use the kubectl uncordon command to mark it as schedulable again. This allows new pods to be scheduled on the replacement node.

---

### 12. Scenario: You have a stateful application running on Kubernetes that requires persistent storage. How would you ensure that the data is retained when the pods are rescheduled or updated?

**Answer:**

To ensure data retention for a stateful application, I would use the following Kubernetes features:

1. Persistent Volumes (PVs) and Persistent Volume Claims (PVCs): I would create a Persistent Volume that represents the storage resource (e.g., a network-attached disk) and then create a Persistent Volume Claim that binds to the PV. This ensures that the same volume is attached to the pod when it's rescheduled or updated.

2. StatefulSets: I would use StatefulSets to manage the stateful application. StatefulSets ensure that each pod has a unique identity and stable network identity, allowing the pod to retain its storage even during rescheduling or

updates. StatefulSets can use PVCs to attach the appropriate Persistent Volumes to each pod.

3. Storage Classes: I would define Storage Classes to dynamically provision Persistent Volumes based on predefined storage requirements. This allows for automated volume provisioning when new PVCs are created. By leveraging these features, the stateful application can maintain its data even when pods are rescheduled, updated, or scaled up/down.

---

### 13. Scenario: A Kubernetes pod is stuck in a "Pending" state. What could be the possible reasons, and how would you troubleshoot it?

**Answer:**

Possible reasons for a pod being stuck in the "Pending" state could include:

1. Insufficient resources: Check if the cluster has enough resources (CPU, memory, storage) to accommodate the pod. You can use the `kubectl describe pod <pod-name>` command to view detailed information about the pod, including any resource-related issues.
2. Unschedulable nodes: Check if all the nodes in the cluster are in the "Ready" state and can schedule the pod. You can use the `kubectl get nodes` command to see the node status.
3. Pod scheduling constraints: Verify if the pod has any scheduling constraints or affinity/anti-affinity rules that are preventing it from being scheduled. Check the pod's YAML or manifest file for any such specifications.
4. Persistent Volume (PV) availability: If the pod requires a Persistent Volume, ensure that the required storage is available and accessible.
5. Network-related issues: Check if there are any network restrictions or misconfigurations preventing the pod from being scheduled or communicating with other resources.

---

### 14. Scenario: You have a Kubernetes Deployment with multiple replicas, and some pods arefailing health checks. How would you identify the root cause and fix it?

**Answer:**

To identify the root cause and fix failing health checks for pods in a Kubernetes Deployment:

1. Check the pod's logs: Use the `kubectl logs <pod-name>` command to retrieve the logs of the failing pod. Inspect the logs for any error messages or exceptions that could indicate the cause of the failure.
2. Verify health check configurations: Examine the readiness and liveness probe configurations in the Deployment's YAML or manifest file. Ensure that the endpoints being probed are correct, the expected response is received, and the success criteria are appropriately defined.
3. Debug container startup: If the pods are failing to start, check the container's startup commands, entrypoints, or initialization processes. Use the `kubectl describe pod <pod-name>` command to get detailed information about the pod, including any container-related errors.

4. Resource constraints: Inspect the resource requests and limits for the pods. It's possible that the pods are exceeding the allocated resources, causing failures. Adjust the resource specifications as necessary.
5. Image issues: Verify that the Docker image being used is correct and accessible. Ensure that the image's version, registry, and repository details are accurate.
6. Rollout issues: If the pods were recently deployed or updated, ensure that the rollout process completed successfully. Check the deployment's status using `kubectl rollout status <deployment-name>` and examine any rollout history with `kubectl rollout history <deployment-name>`.

---

### 15. Scenario: You need to scale a Kubernetes Deployment manually. How would you accomplish this?

**Answer:**

To manually scale a Kubernetes Deployment:

- Use the `kubectl scale` command: Run `kubectl scale deployment/<deployment-name> --replicas=<number-of-replicas>` to scale the deployment. Replace `<deployment-name>` with the name of your deployment, and `<number-of-replicas>` with the desired number of replicas.
- Alternatively, update the Deployment YAML: Modify the `replicas` field in the Deployment's YAML or manifest file to the desired number of replicas. Then, apply the changes using `kubectl apply -f <path-to-deployment-yaml>`.