Certified Kubernetes Application Developer (CKAD)

By Damian Igbe, PhD

Day 2

Domain 1.2: Choose and Use the Right Workload Resource (Deployment, DaemonSet, CronJob, etc.)

In Kubernetes, workloads are the resources responsible for running your applications. They define the application's deployment and execution patterns, scaling, and availability. As a Kubernetes Application Developer (CKAD), one of your key responsibilities is to choose the appropriate workload resource for your application based on your requirements.
In this section of the CKAD training, we will explore the different types of workload resources available in Kubernetes and how to use them effectively.

1. What are Workload Resources in Kubernetes?

A workload resource in Kubernetes is a resource type that helps you run your application containers in a Kubernetes cluster. It defines the desired state for your application (e.g., how many replicas of a container to run, whether it should be scheduled on specific nodes, etc.) and allows Kubernetes to manage the lifecycle of the application according to that state. There are several types of workload resources in Kubernetes, and each has specific use cases. Some of the most common workload resources are:

- Deployment
- DaemonSet
- StatefulSet
- ReplicaSet
- Job
- CronJob

2. Deployments: Managing Stateless Applications

A Deployment is the most common workload resource used in Kubernetes for stateless applications. It provides declarative updates to applications, meaning you can describe the desired state of your application, and Kubernetes will automatically manage the deployment to match that state.

Use Cases:

1. Stateless applications where replicas of a pod can be spread across nodes for load balancing.
2. Rolling updates and rollbacks for zero-downtime deployment.

Key Features:

1. Scaling: Automatically adjusts the number of replicas as required.
2. Rolling Updates: Kubernetes automatically rolls out updates to your application without downtime.
3. Rollback: If something goes wrong with the new deployment, you can easily roll back to the previous version.

Example: To create a simple Deployment for a web application, you can use the following YAML:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-web-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-web-app
  template:
    metadata:
      labels:
        app: my-web-app
    spec:
      containers:
        - name: my-web-container
          image: nginx
          ports:
            - containerPort: 80
```

This YAML file describes a deployment with three replicas of the my-web-app container, ensuring high availability.

Use Kubernetes Primitives to Implement Common Deployment Strategies (e.g., Blue/Green or Canary)

In Kubernetes, deploying applications efficiently and safely is critical to maintaining high availability and minimizing downtime. One of the primary ways to achieve this is by using deployment strategies such as Blue/Green and Canary deployments. These strategies allow you to roll out updates incrementally, minimizing risk and enabling better control over the release process.
As a Kubernetes Application Developer (CKAD), you must understand how to implement these strategies using Kubernetes primitives such as Deployments, Services, Labels, and Selectors.

1. Blue/Green Deployment Strategy

Blue/Green Deployment is a deployment strategy where two environments (the "Blue" and "Green" environments) are maintained. The "Blue" environment is the live, production version of your application, while the "Green" environment contains the updated version of the application.

How it Works:

1. Initially, the "Blue" environment (the live version) serves all the traffic.
2. The "Green" environment is deployed and tested in parallel with the "Blue" environment.
3. Once the "Green" environment is tested and verified, the traffic is switched to the "Green" environment, making it the new production.
4. The "Blue" environment can be kept as a fallback in case of issues with the new deployment.

Advantages:

1. Minimized risk: If there is an issue with the "Green" deployment, you can quickly revert to the "Blue" version.
2. Zero downtime: With traffic routed between two separate environments, you can achieve a seamless switch.

Implementation in Kubernetes:

To implement Blue/Green in Kubernetes, you can use two separate Deployments (one for Blue, one for Green) and a Service that directs traffic to the active environment.

Step 1: Define Blue and Green Deployments

Blue Deployment (Current live version)

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

```
    matchLabels:
      app: my-app
      version: blue
  template:
    metadata:
      labels:
        app: my-app
        version: blue
    spec:
      containers:
        - name: my-app
          image: my-app:blue
          ports:
            - containerPort: 80
```

Green Deployment (New version to be deployed)

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-green
spec:
  replicas: 0  # Initially set to 0 replicas until switch
  selector:
    matchLabels:
      app: my-app
      version: green
  template:
    metadata:
      labels:
        app: my-app
        version: green
    spec:
      containers:
        - name: my-app
          image: my-app:green
          ports:
            - containerPort: 80
```

Step 2: Define a Service to Route Traffic

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
```

```
  selector:
    app: my-app
    version: blue  # Initially routes to the blue version
  ports:
    - port: 80
      targetPort: 80
```

Step 3: Switch Traffic to the Green Deployment

Once you have tested the "Green" deployment, you can modify the Service to point to the new version:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
    version: green  # Switch traffic to green version
  ports:
    - port: 80
      targetPort: 80
```

Step 4: Scale Down/Remove Blue Deployment (Optional)

After the traffic is fully switched to the "Green" environment, you can scale down or delete the "Blue" Deployment:

```
 kubectl scale deployment app-blue --replicas=0
```

2. Canary Deployment Strategy

Canary Deployment is a strategy where a new version of the application is deployed to a small subset of users before being rolled out to the entire user base. This allows you to monitor the performance of the new version in a production environment with minimal risk.

How it Works:

1. A small portion of traffic is routed to the new version (the "canary").
2. The rest of the traffic continues to go to the stable version.
3. If the new version performs well (i.e., no critical issues are found), more traffic is gradually shifted towards it until it is fully deployed.
4. If issues are detected, the new version can be rolled back or adjusted.

Advantages:

- Incremental rollout: Allows testing in a real-world environment with minimal risk.
- Easy rollback: If something goes wrong, the rollout can be stopped, and the traffic can be directed back to the stable version.

Implementation in Kubernetes:

To implement Canary deployment in Kubernetes, you typically use a Deployment and incrementally change the number of replicas for the new version. This is managed by adjusting the replica count for both the stable and canary versions.

Step 1: Define the Stable Version (e.g., stable) and Canary Version (e.g., canary) Deployments

Stable Version Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-stable
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
      version: stable
  template:
    metadata:
      labels:
        app: my-app
        version: stable
    spec:
      containers:
        - name: my-app
          image: my-app:stable
          ports:
            - containerPort: 80
```

Canary Version Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: app-canary
spec:
  replicas: 1  # Start with a small number of replicas for the canary
```

```
  selector:
    matchLabels:
      app: my-app
      version: canary
  template:
    metadata:
      labels:
        app: my-app
        version: canary
    spec:
      containers:
        - name: my-app
          image: my-app:canary
          ports:
            - containerPort: 80
```

Step 2: Define a Service

The Service will initially route traffic to both the stable and canary versions. As the rollout progresses, you can gradually change the traffic distribution.

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

Step 3: Gradually Increase the Canary Replicas

Initially, the canary version will have a small number of replicas (e.g., 1 replica). Over time, you can gradually scale up the canary deployment and scale down the stable version to increase traffic to the new version:

```
kubectl scale deployment app-canary --replicas=3  # Increase canary replicas

kubectl scale deployment app-stable --replicas=2  # Scale down stable version
```

Step 4: Monitor and Adjust

While the canary deployment is running, monitor its performance. If it's working as expected, continue scaling it up. If any issues arise, you can easily scale it back down or roll it back to the stable version.

Step 5: Complete the Deployment

Once the canary version is fully tested, you can scale it to the desired number of replicas, ensuring that all traffic is now routed to the new version.

3. Comparison of Blue/Green and Canary Deployment

Feature	Blue/Green Deployment	Canary Deployment
Traffic Management	Entire traffic switches from Blue to Green once the new version is ready.	Gradually shifts a small portion of traffic to the new version.
Rollback	Rollback to the Blue version is instantaneous.	Rollback can be done by scaling down the canary replicas.
Risk Level	Low risk once the Green environment is validated.	Low risk with gradual exposure of the new version.
Deployment Complexity	Simpler, but requires managing two complete environments.	More complex, requires monitoring and gradual rollout.
Use Case	Ideal for major version upgrades with minimal downtime.	Ideal for testing a new version with a subset of users.

4. Conclusion

As a Kubernetes Application Developer (CKAD), understanding how to implement Blue/Green and Canary deployments will empower you to release applications safely and with minimal risk. These strategies help you manage application updates in a controlled manner, ensuring high availability and reducing the impact of issues that may arise with new versions.
Both strategies use Kubernetes primitives like Deployments, Services, Labels, and Selectors, allowing for flexibility in managing how updates are rolled out across your cluster.
Blue/Green deployments are excellent for switching between major versions with minimal downtime. Canary deployments are useful for incrementally rolling out updates and monitoring their impact on production environments. By mastering these strategies, you can ensure more resilient, reliable, and efficient application rollouts in Kubernetes.

Understand How to Perform Rolling Updates with Deployments

1. **Understanding Rolling Updates**

Rolling updates allow you to update an application without downtime by gradually replacing old pods with new ones. This process ensures that some instances of the application remain available while new versions are deployed.

How Rolling Updates Work:

Step 1: When you update the Deployment (e.g., change the Docker image), Kubernetes will create new pods with the updated configuration.

Step 2: It will slowly scale down the old pods and scale up the new ones, maintaining the desired number of pods running throughout the process.

Step 3: The new version of the pods will only be rolled out if they pass health checks, preventing the deployment of a failing version.

Step 4: If necessary, the update can be rolled back to a previous stable version.

Advantages of Rolling Updates:

1. Zero Downtime: Rolling updates help ensure that at least some replicas of the application are always running, preventing downtime during updates.
2. Safe Gradual Rollout: If issues occur with the new version, you can pause or roll back the update.
3. Version Control: You can easily revert to a previous stable version if necessary.

2. Creating a Deployment

To use a Deployment in Kubernetes, you create a YAML manifest that defines the deployment's configuration, including the Docker image, number of replicas, and other parameters.

Example of a simple Deployment manifest for an application:

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

```
  metadata:
    labels:
      app: my-app
  spec:
    containers:
      - name: my-app-container
        image: my-app:v1  # Version 1 of the app
        ports:
          - containerPort: 80
```

In this example:

1. The deployment is named my-app-deployment.
2. Three replicas are defined, meaning Kubernetes will maintain three pods running at all times.
3. The image: my-app:v1 defines the version of the Docker image being used for the application.

3. Performing a Rolling Update

Kubernetes automatically performs a rolling update when you change the Deployment's specification, such as updating the image version. The update can be done via the command line (kubectl) or by modifying the Deployment YAML file.

Step-by-Step Guide to Perform a Rolling Update:

Update the Deployment with a New Docker Image: You can modify the Deployment by changing the Docker image tag (e.g., from v1 to v2). The rolling update will be initiated when the Deployment is updated.

Example: You can replace the image with nginx.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app-container
          image: my-app:v2  # Updated to version 2 of the app
          ports:
            - containerPort: 80
```

To apply the update, run:

kubectl apply -f my-app-deployment.yaml

Alternatively, you can update the image directly using kubectl set image:

kubectl set image deployment/my-app-deployment my-app-container=my-app:v2

Kubernetes Starts the Rolling Update:

Kubernetes will replace the old pods (running v1) with new pods (running v2) one by one.

- It ensures that the desired number of replicas (3 in this case) are running throughout the update process.
- It monitors the health of the new pods (via liveness and readiness probes) before scaling down the old ones.

Verify the Rolling Update: You can track the progress of the rolling update with the following command:

kubectl rollout status deployment/my-app-deployment

This will show the status of the rolling update, such as "deployment rolling update complete."

Rollback if Necessary: If the rolling update encounters issues, Kubernetes can roll back to the previous stable version of the deployment. This can be done with the following command:

kubectl rollout undo deployment/my-app-deployment

This command will restore the previous version of the deployment and revert the changes.

4. Customizing Rolling Update Behavior

Kubernetes provides several options for customizing the behavior of rolling updates. You can define how quickly the update proceeds, how many pods can be unavailable at a time, and other parameters. These options are specified in the Deployment spec under the rollingUpdate strategy.

Example of Customizing Rolling Update Strategy:

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

```
replicas: 3
selector:
  matchLabels:
    app: my-app
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1       # Maximum number of pods that can be created above the desired number of pods
    maxUnavailable: 1  # Maximum number of pods that can be unavailable during the update
template:
  metadata:
    labels:
      app: my-app
  spec:
    containers:
      - name: my-app-container
        image: my-app:v2  # Updated image version
        ports:
          - containerPort: 80
```

In this example:

1. maxSurge: 1 allows one additional pod to be created during the update, ensuring that the overall pod count can temporarily exceed the desired replicas.
2. maxUnavailable: 1 ensures that no more than one pod can be unavailable at any time during the update.

5. Troubleshooting Rolling Updates

Occasionally, rolling updates can encounter issues, such as pods failing health checks or misconfigured resources. Here are some common troubleshooting steps:

Check Deployment Status: This command provides information about the current state of the update, whether it's progressing or has stalled.

kubectl rollout status deployment/my-app-deployment

Check Pod Logs. If the new pods aren't starting correctly, check their logs for errors.

kubectl logs <pod-name>

Revert to the Previous Version: to roll back to the previous stable version use

kubectl rollout undo

6. Conclusion

As a Kubernetes Application Developer (CKAD), understanding how to configure and manage Deployments and perform Rolling Updates is essential to deploying applications safely and efficiently. Rolling updates allow you to update applications with minimal disruption and no downtime, which is crucial for production-grade environments.

1. Deployments are used to manage the lifecycle of your applications and ensure that the desired state is always maintained.
2. Rolling updates enable you to gradually replace old versions of an application with new ones, while ensuring that the service remains available.
3. Customizing rolling update parameters like maxSurge and maxUnavailable helps fine-tune the update process for your specific needs.

By mastering these concepts, you'll be able to implement seamless and safe application updates in your Kubernetes clusters.

2. Managing Stateful Applications with StatefulSets

A StatefulSet is used for stateful applications that require unique network identifiers, stable storage, and ordered deployment. It is ideal for applications like databases or distributed file systems that need persistent state across pod restarts.
Use Cases:

- Stateful applications such as databases (e.g., MySQL, MongoDB), message queues (e.g., Kafka), or distributed file systems (e.g., GlusterFS).
- Applications that require stable, persistent storage and network identity.

  Key Features:

  Stable, unique pod names: Each pod in a StatefulSet gets a unique, stable name (e.g., my-app-0, my-app-1).

  1. Persistent storage: Allows each pod to be associated with a persistent volume claim (PVC).
  2. Ordered deployment and scaling: Pods are created and terminated in a specific order.

     Example: A StatefulSet for running a stateful web application:

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: my-db
spec:
  serviceName: "my-db"
  replicas: 3
  selector:
    matchLabels:
      app: my-db
  template:
    metadata:
      labels:
        app: my-db
    spec:
      containers:
        - name: my-db-container
          image: mysql:5.7
          volumeMounts:
            - name: db-data
              mountPath: /data/db
  volumeClaimTemplates:
    - metadata:
        name: db-data
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests:
            storage: 1Gi
```

This example ensures that each pod in the StatefulSet has its own persistent storage for database data.

LAB walkthrough: Create a StatefulSet

PVCs and PVs can be automatically created based on VolumeClaimTemplates.

Note volumeClaimTemplates represents a type of template that the system uses to create PVCs. The number of PVCs equals the number of replicas that are deployed for the StatefulSet application. The configurations of these PVCs are the same except for the PVC names.

1. Use the following template to create a file named statefulset.yaml.

Deploy a Service and a StatefulSet, and provision two pods for the StatefulSet.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx
  labels:
```

```
    app: nginx
spec:
  ports:
  - port: 80
    name: web
  clusterIP: None
  selector:
    app: nginx
---
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: web
spec:
  selector:
    matchLabels:
      app: nginx
  serviceName: "nginx"
  replicas: 2
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx
        ports:
        - containerPort: 80
          name: web
        volumeMounts:
        - name: disk-ssd
          mountPath: /data
  volumeClaimTemplates:
  - metadata:
      name: disk-ssd
    spec:
      accessModes: [ "ReadWriteOnce" ]
      storageClassName: "standard"
      resources:
        requests:
          storage: 2Gi
```

- replicas: the parameter is set to 2 in this example. This indicates that two pods are created.
- mountPath: the path where you want to mount the disk in the container.
- accessModes: the access mode of the StatefulSet.

- storageClassName: the parameter is set to standard in this example. This indicates that a standard storage class was created and used.
- storage: specifies the storage capacity that is required by the application.

2. Run the following command to create a StatefulSet:

```
kubectl create -f statefulset.yaml
```

3. Run the following command to query the deployed pods:

```
kubectl get pod
```

Expected output:

```
NAME                READY   STATUS    RESTARTS   AGE
web-0               1/1     Running   0          6m
web-1               1/1     Running   0          6m
```

4. Run the following command to query the PVCs:

```
kubectl get pvc
```

Expected output:

```
NAME           STATUS   VOLUME                   CAPACITY   ACCESS MODES   STORAGECLASS       AGE
disk-ssd-web-0 Bound    d-2zegw7et6xc96nbojuoo   2Gi        RWO            alicloud-disk-ssd  7m
disk-ssd-web-1 Bound    d-2zefbrqggvkd10xb523h   2Gi        RWO            alicloud-disk-ssd  6m
```

Verify that the PVCs are scaled out together with the StatefulSet application

1. Run the following command to scale out the StatefulSet application to three pods:

```
kubectl scale sts web --replicas=3
```

Expected output:

```
statefulset.apps/web scaled
```

2. Run the following command to view the pods after the StatefulSet application is scaled out:

kubectl get pod

Expected output:

NAME	READY	STATUS	RESTARTS	AGE
web-0	1/1	Running	0	34m
web-1	1/1	Running	0	33m
web-2	1/1	Running	0	26m

3. Run the following command to view the PVCs after the StatefulSet application is scaled out:

kubectl get pvc

Expected output:

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
disk-ssd-web-0	Bound	d-2zegw7et6xc96nbojuoo	2Gi	RWO	standard	35m
disk-ssd-web-1	Bound	d-2zefbrqggvkd10xb523h	2Gi	RWO	standard	34m
disk-ssd-web-2	Bound	d-2ze4jx1zymn4n9j3pic2	2Gi	RWO	standard	27m

The output indicates that three PVCs are provisioned for the StatefulSet application after the StatefulSet application is scaled to three pods.

Verify that the PVCs remain unchanged after the StatefulSet application is scaled in.

1. Run the following command to scale the StatefulSet application to two pods:

kubectl scale sts web --replicas=2

Expected output:
statefulset.apps/web scaled

2. Run the following command to view the pods after the StatefulSet application is scaled in:

kubectl get pod

Expected output:

```
NAME                READY   STATUS    RESTARTS   AGE
web-0               1/1     Running   0          38m
web-1               1/1     Running   0          38m
```

Only two pods are deployed for the StatefulSet application.

3. Run the following command to view the PVCs after the StatefulSet application is scaled in:

```
kubectl get pvc
```

Expected output:

```
NAME          STATUS   VOLUME                CAPACITY   ACCESS MODES   STORAGECLASS       AGE
disk-ssd-web-0   Bound    d-2zegw7et6xc96nbojuoo   2Gi        RWO            standard           39m
disk-ssd-web-1   Bound    d-2zefbrqggvkd10xb523h   2Gi        RWO            standard           39m
disk-ssd-web-2   Bound    d-2ze4jx1zymn4ngj3pic2   2Gi        RWO            standard           39m
```

After the StatefulSet application is scaled to two pods, the StatefulSet application still has three PVCs. This indicates that the PVCs are not scaled in together with the StatefulSet application.

Verify that the PVCs remain unchanged when the StatefulSet application is scaled out again

When the StatefulSet is scaled out again, verify that the PVCs remain unchanged.

1. Run the following command to scale out the StatefulSet application to three pods:

```
kubectl scale sts web --replicas=3
```

Expected output:

```
statefulset.apps/web scaled
```

2. Run the following command to view the pods after the StatefulSet application is scaled out:

```
kubectl get pod
```

Expected output:

```
NAME                 READY   STATUS    RESTARTS   AGE
web-0                1/1     Running   0          1h
web-1                1/1     Running   0          1h
web-2                1/1     Running   0          8s
```

3. Run the following command to view the PVCs after the StatefulSet application is scaled out:

```
kubectl get pvc
```

Expected output:

```
NAME          STATUS   VOLUME                CAPACITY   ACCESS MODES   STORAGECLASS       AGE
disk-ssd-web-0   Bound    d-2zegw7et6xc96nbojuoo     2Gi        RWO            standard           50m
disk-ssd-web-1   Bound    d-2zefbrqggvkd10xb523h     2Gi        RWO            standard           50m
disk-ssd-web-2   Bound    d-2ze4jx1zymn4ng3pic2      2Gi        RWO            standard           50m
```

The newly created pod uses an existing PVC.

Verify that the PVCs remain unchanged when the pod of the StatefulSet application is deleted

1. Run the following command to view the PVC that is used by the pod named web-1:

```
kubectl describe pod web-1 | grep ClaimName
```

Expected output:

```
ClaimName:  disk-ssd-web-1
```

2. Run the following command to delete the pod named web-1:

```
kubectl delete pod web-1
```

Expected output:

pod "web-1" deleted

3. Run the following command to view the pod:

kubectl get pod

Expected output:

NAME	READY	STATUS	RESTARTS	AGE
web-0	1/1	Running	0	1h
web-1	1/1	Running	0	25s
web-2	1/1	Running	0	9m

The recreated pod uses the same name as the deleted pod.

4. Run the following command to view the PVCs:

kubectl get pvc

Expected output:

NAME	STATUS	VOLUME	CAPACITY	ACCESS MODES	STORAGECLASS	AGE
disk-ssd-web-0	Bound	d-2zegw7et6xc96nbojuoo	2Gi	RWO	standard	1h
disk-ssd-web-1	Bound	d-2zefbrqggvkd10xb523h	2Gi	RWO	standard	1h
disk-ssd-web-2	Bound	d-2ze4jx1zymn4n9j3pic2	2Gi	RWO	standard	1h

The recreated pod uses an existing PVC.

Verify that the StatefulSet application supports persistent storage

1. Run the following command to view the file in the /data path:

kubectl exec web-1 -- ls /data

Expected output:
lost+found

2. Run the following command to create a file named statefulset in the /data path:

```
kubectl exec web-1 -- touch /data/statefulset
```

3. Run the following command to view the file in the /data path:

```
kubectl exec web-1 -- ls /data
```

Expected output:
lost+found

statefulset

4. Run the following command to delete the pod named web-1:

```
kubectl delete pod web-1
```

Expected output:
pod "web-1" deleted

5. Run the following command to view the file in the /data path:

```
kubectl exec web-1 -- ls /data
```

Expected output:
lost+found

statefulset

The statefulset file still exists in the /data path. This indicates that data is persisted to the disk.

3. DaemonSets: Running One Pod per Node

A DaemonSet ensures that a copy of a specific pod is running on all (or some) nodes in the cluster. This is particularly useful for node-level services such as monitoring agents, log collectors, or networking tools that need to run on each node.

Use Cases:

1. Cluster-wide services that need to be run on every node (e.g., logging agents, monitoring agents).
2. Single-application per node deployment.

Key Features:

1. Ensures that each node gets a pod (or a set of pods).
2. Pods can be scheduled only on certain nodes by using node selectors, affinities, or taints and tolerations.
3. Automatically adds new pods to newly added nodes to the cluster.

Example: To run a DaemonSet that deploys a logging agent on every node:

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: fluentd
spec:
  selector:
    matchLabels:
      app: fluentd
  template:
    metadata:
      labels:
        app: fluentd
    spec:
      containers:
        - name: fluentd
          image: fluent/fluentd:v1.12-debian-1
          volumeMounts:
            - name: varlog
              mountPath: /var/log
      volumes:
        - name: varlog
          hostPath:
            path: /var/log
```

This example runs a logging agent (fluentd) on each node, ensuring that logs are collected across all nodes in the cluster.

4. CronJobs: Running Jobs on a Schedule

A CronJob is a workload resource that runs jobs on a scheduled basis, similar to a cron job in Unix-based systems. This is useful for tasks that need to be executed periodically, such as backups, report generation, or cleanup tasks.

Use Cases:

1. Periodic tasks like backups, database cleanups, or scheduled data processing.
2. Batch jobs that need to run at specific times or intervals.

Key Features:

1. Cron-like syntax for specifying the schedule (e.g., 0 0 * * * to run at midnight every day).
2. Job management: Automatically creates a new job at each scheduled time and ensures that the job completes successfully.
3. Concurrency policies: Control whether multiple jobs can run concurrently or if the next job should wait for the previous one to finish.

Example 1

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
          - name: hello
            image: busybox
            args:
            - /bin/sh
            - -c
            - date; echo Hello from the Kubernetes cluster
          restartPolicy: OnFailure
```

Example 2: A CronJob that runs a backup every day at midnight:

```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: backup-cronjob
spec:
  schedule: "0 0 * * *"  # Run at midnight every day
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: nginx
              image: backup-image
              command: ["/bin/sh", "-c", "backup.sh"]
          restartPolicy: OnFailure
```

This cron job runs a backup script every day at midnight.

5. Jobs: Running One-Time Tasks

A Job is a Kubernetes workload used for one-time, batch processing tasks that can be completed in the background, such as processing a queue, migrating data, or running a report.

Use Cases:

1. One-off tasks like database migrations or data processing.
2. Batch jobs that can run to completion and do not need to persist.

Key Features:

1. Completion tracking: Jobs are considered complete when the specified number of pods successfully terminate.
2. Parallel jobs: You can run multiple jobs concurrently or sequentially depending on your use case.

Example: A Job for running a database migration:

```
apiVersion: batch/v1
kind: Job
metadata:
  name: jobhello
spec:
  template:
    spec:
      containers:
      - name: jobhello
        image: busybox
        args:
        - /bin/sh
        - -c
        - date; echo Hello from the Kubernetes cluster
      restartPolicy: OnFailure
```

6. Conclusion: Choosing the Right Workload Resource

As a Kubernetes Application Developer (CKAD), choosing the correct workload resource for your application is a critical skill. Here's a summary of when to use each type:

- Deployment: For stateless applications that need scaling and rolling updates.
- DaemonSet: For running a pod on every node (e.g., monitoring, logging).
- StatefulSet: For stateful applications requiring stable storage and identity.
- CronJob: For running jobs on a scheduled basis (e.g., backups, periodic tasks).
- Job: For one-off tasks that run to completion.

Selecting the correct workload resource helps optimize application performance, scalability, and maintainability, and ensures that your applications run efficiently in a Kubernetes cluster.

Domain 2.3: Use the Helm Package Manager to Deploy Existing Packages

Helm is the most widely used package manager for Kubernetes applications. It simplifies the process of deploying, configuring, and managing Kubernetes resources by enabling you to use Helm charts, which are pre-configured Kubernetes manifests bundled together for easy installation and management. As a Kubernetes Application Developer (CKAD), mastering Helm will help you streamline your deployment process and simplify the management of complex applications.

In this section, we'll cover how to use Helm to deploy existing Helm charts (pre-packaged applications) on a Kubernetes cluster.

1. What is Helm?

Helm is a tool that helps you manage Kubernetes applications through Helm charts. A Helm chart is a collection of files that describe a related set of Kubernetes resources, such as Deployments, Services, ConfigMaps, and more. Helm packages these resources together, allowing you to deploy and manage them as a single unit.

Key Concepts:

1. Helm Chart: A package of pre-configured Kubernetes resources.
2. Release: An instance of a Helm chart deployed to a Kubernetes cluster.
3. Repository: A location where Helm charts are stored and shared (e.g., Helm Hub, Artifact Hub).
4. Helm Client: The command-line tool used to interact with Helm charts and repositories.

2. Installing Helm

Before using Helm to deploy packages, you need to install it on your local machine. Here's how you can install Helm:

Install Helm (on macOS/Linux):

On macOS using Homebrew

brew install helm

On Linux (via script)

curl https://raw.githubusercontent.com/helm/helm/master/scripts/get-helm-3 | bash

Install Helm (on Windows):

You can use Chocolatey or Windows Subsystem for Linux (WSL) to install Helm on Windows:

choco install kubernetes-helm

or

winget install helm

3. Helm Repositories

Helm charts are stored in repositories, which are collections of Helm charts. By default, Helm includes the Helm Hub repository, but you can add your own or use third-party repositories.

Add a Helm Repository:

To add a new repository (e.g., the official stable repository), use the following command:

```
helm repo add stable https://charts.helm.sh/stable
```

Once added, you can update your local Helm chart repository cache:

```
helm repo update
```

4. Deploying Existing Helm Charts

To deploy an existing Helm chart to your Kubernetes cluster, you can follow these steps. We'll use the nginx-ingress chart from the official Helm repository as an example.

Step-by-Step:

1. Search for Helm Charts: To search for a chart in a Helm repository, use the helm search command:

```
helm search repo nginx-ingress
```

This command will return a list of charts related to nginx-ingress available in your Helm repositories.

2. Install the Helm Chart:

To deploy the chart, use the helm install command followed by the release name (an identifier for the deployed application) and the name of the chart.

```
helm install my-ingress stable/nginx-ingress
```

In this example:

- my-ingress is the release name.
- stable/nginx-ingress is the Helm chart you're installing from the stable repository.
- Helm will automatically create the necessary Kubernetes resources (Deployments, Services, ConfigMaps, etc.) based on the chart and deploy them to your Kubernetes cluster.

3. Verify the Deployment:

After installation, you can check the status of your release:

helm list

This will show you all the releases installed in your cluster.
You can also check the individual Kubernetes resources created by the chart, such as Pods and Services:

kubectl get pods

kubectl get services

4. Check the Resources Created by the Helm Chart:

To view the Kubernetes resources created by Helm, use the following command:

helm get all my-ingress

This will show you detailed information about the release, including the manifests of the deployed resources.

Install nginx from bitnami

helm repo add bitnami1 https://charts.bitnami.com/bitnam

helm search repo bitnami | grep nginx

helm install bitnami/nginx --generate-name #or helm install nginx bitnami/nginx

Expected Output:

NAME: nginx
LAST DEPLOYED: Mon Feb 24 21:43:42 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: nginx
CHART VERSION: 19.0.0
APP VERSION: 1.27.4
Did you know there are enterprise versions of the Bitnami catalog? For enhanced secure software supply chain features, unlimited pulls from Docker, LTS support, or application customization, see Bitnami Premium or Tanzu Application Catalog. See https://www.arrow.com/globalecs/na/vendors/bitnami for more information.
** Please be patient while the chart is being deployed **
NGINX can be accessed through the following DNS name from within your cluster:
nginx.default.svc.cluster.local (port 80)

To access NGINX from outside the cluster, follow the steps below:
1. Get the NGINX URL by running these commands:
  NOTE: It may take a few minutes for the LoadBalancer IP to be available.
       Watch the status with: 'kubectl get svc --namespace default -w nginx'
   export SERVICE_PORT=$(kubectl get --namespace default -o jsonpath="{.spec.ports[0].port}" services nginx)
   export SERVICE_IP=$(kubectl get svc --namespace default nginx -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
   echo "http://${SERVICE_IP}:${SERVICE_PORT}"
WARNING: There are "resources" sections in the chart not set. Using "resourcesPreset" is not recommended for production. For production installations, please set the following values according to your workload needs:
  - cloneStaticSiteFromGit.gitSync.resources
  - resources
+info https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
⚠ SECURITY WARNING: Original containers have been substituted. This Helm chart was designed, tested, and validated on multiple platforms using a specific set of Bitnami and Tanzu Application Catalog containers. Substituting other containers is likely to cause degraded security and performance, broken chart features, and missing environment variables.
Substituted images detected:
  - docker.io/bitnami/nginx:1.27.4-debian-12-r1
⚠ WARNING: Original containers have been retagged. Please note this Helm chart was tested, and validated on multiple platforms using a specific set of Tanzu Application Catalog containers. Substituting original image tags could cause unexpected behavior.
Retagged images:
  - docker.io/bitnami/nginx:1.27.4-debian-12-r1

Check your deployments:

helm list

kubectl get deploy

kubectl get pods

kubec get svc

kubectl get ep

Install mysql from Bitnami

helm install mysql bitnami/mysql

Expected output:

NAME: mysql
LAST DEPLOYED: Mon Feb 24 21:48:57 2025
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
NOTES:
CHART NAME: mysql
CHART VERSION: 12.2.2
APP VERSION: 8.4.4

Did you know there are enterprise versions of the Bitnami catalog? For enhanced secure software supply chain features, unlimited pulls from Docker, LTS support, or application customization, see Bitnami Premium or Tanzu Application Catalog. See https://www.arrow.com/globalecs/na/vendors/bitnami for more information.
** Please be patient while the chart is being deployed **
Tip:
  Watch the deployment status using the command: kubectl get pods -w --namespace default
Services:
  echo Primary: mysql.default.svc.cluster.local:3306
Execute the following to get the administrator credentials:
  echo Username: root
  MYSQL_ROOT_PASSWORD=$(kubectl get secret --namespace default mysql -o jsonpath="{.data.mysql-root-password}" | base64 -d)
To connect to your database:
  1. Run a pod that you can use as a client:
kubectl run mysql-client --rm --tty -i --restart='Never' --image docker.io/bitnami/mysql:8.4.4-debian-12-r0 --namespace default --env MYSQL_ROOT_PASSWORD=$MYSQL_ROOT_PASSWORD --command -- bash
  2. To connect to primary service (read/write):
      mysql -h mysql.default.svc.cluster.local -uroot -p"$MYSQL_ROOT_PASSWORD"
WARNING: There are "resources" sections in the chart not set. Using "resourcesPreset" is not recommended for production. For production installations, please set the following values according to your workload needs:
  - primary.resources
  - secondary.resources
+info https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
https://github.com/bitnami/charts/tree/main/bitnami/nginx/#installing-the-chart

5. Customizing the Helm Chart During Deployment

Helm allows you to customize the deployment of a Helm chart by overriding its default values using a values.yaml file or command-line flags. Using

--set Flag:

You can modify specific values directly through the --set flag:

helm install ingress stable/nginx-ingress --set replicaCount=3

This will override the default replica count to 3 instead of the default value specified in the chart.

You can see variables you can control from the files itself at:

https://github.com/bitnami/charts/blob/main/bitnami/nginx-ingress-controller/values.yaml

https://github.com/bitnami/charts/tree/main/bitnami/nginx-ingress-controller

https://bitnami.com/stack/nginx-ingress-controller/helm

Using a Custom values.yaml File:

Alternatively, you can create a custom values.yaml file and provide it during installation:
controller:

replicaCount: 3

Then, install the chart with your custom values:

helm install my-ingress stable/nginx-ingress -f custom-values.yaml

This approach is useful when you need to provide more complex configuration changes.

6. Upgrading an Existing Release

If you want to upgrade an existing Helm release (e.g., to a new version of the chart or with new custom values), you can use the helm upgrade command:

helm upgrade my-ingress stable/nginx-ingress --set replicaCount=5

This will update the my-ingress release with the new chart version and custom values.

7. Rolling Back a Release

If an upgrade or configuration change causes issues, you can roll back to a previous version of the release using the helm rollback command:

helm rollback my-ingress 1

In this example, 1 refers to the revision number of the release. You can view all revisions of a release with:

helm history my-ingress

This will show you the history of changes made to the release, including the revision numbers.

8. Uninstalling a Helm Release

To remove a Helm release, you can use the helm uninstall command:

helm uninstall my-ingress

This will delete all Kubernetes resources associated with the my-ingress release.

9. Benefits of Using Helm for Deployments

1. Reusability: Helm charts allow you to easily reuse application templates across multiple projects or clusters.
2. Simplified Management: Helm packages and manages complex Kubernetes applications, making it easier to handle updates, rollbacks, and configuration management.
3. Versioning: Helm supports versioned releases, making it easy to track and roll back application changes.
4. Customization: Helm allows you to easily customize application configurations without modifying the underlying YAML files manually.

10. Conclusion

Using Helm to deploy existing packages is an essential skill for a Kubernetes Application Developer (CKAD). It simplifies application deployment, management, and upgrades, especially for complex applications with multiple Kubernetes resources.
By mastering Helm, you can:

1. Easily deploy pre-configured Kubernetes applications (Helm charts).
2. Customize and manage your deployments using values files or command-line overrides.
3. Simplify updates and rollbacks using Helm's release management capabilities.

With Helm, you can focus on building and developing applications rather than worrying about the complexity of Kubernetes YAML files.

Domain 2.4: Kustomize

Kustomize is a powerful tool built into Kubernetes for customizing Kubernetes resource YAML files. It enables you to manage configurations in a more flexible and reusable manner by allowing you to define a "base" configuration and customize it per environment without modifying the original resource files.

In this section, we will dive into how Kustomize works, how to use it for customizing Kubernetes resources, and why it is essential for a Kubernetes Application Developer (CKAD) to understand it.

1. What is Kustomize?

Kustomize is a tool that helps Kubernetes users manage their configurations more efficiently. Unlike Helm, which uses templates, Kustomize works by creating a customization layer on top of existing Kubernetes manifests. Kustomize allows you to reuse the same base configuration files across different environments (e.g., development, staging, production) with environment-specific overlays.

Key features of Kustomize:

1. No Templating: Kustomize does not require any templating logic (like Helm), making it simpler to understand.
2. Customization: It enables you to customize resource definitions (e.g., replicas, image names, labels, etc.) per environment using overlays.
3. Layered Configuration: You can create a "base" set of configurations and then apply customizations on top of it for specific environments.

2. Kustomize Workflow

The Kustomize workflow revolves around two main concepts:

- Base: The core configuration files (e.g., deployments, services) that are common across all environments.
- Overlay: Environment-specific customizations on top of the base configuration (e.g., changing the number of replicas or the image version for a specific environment).

The overall process looks like this:

1. Create a base configuration with reusable Kubernetes manifests.
2. Define environment-specific overlays that modify the base configuration.
3. Apply the kustomized resources using the kubectl command.

3. Kustomize Directory Structure

A typical Kustomize project might have the following directory structure:

my-app

├── base

│ ├── deployment.yaml

│ ├── kustomization.yaml

```
|   └── service.yaml
└── overlay
    ├── dev
    |   ├── deployment.yaml
    |   └── kustomization.yaml
    └── prod
        ├── deployment.yaml
        └── kustomization.yaml
```

1. Base Directory: Contains the common resources that will be reused across all environments.
2. Overlays Directory: Contains the environment-specific customizations.
3. kustomization.yaml File: This file is used by Kustomize to know which resources and customizations to apply.

4. Using Kustomize: Basic Example

Let's walk through a simple example of how to use Kustomize to customize Kubernetes manifests.

Step 1: Create Base Configuration

In your base/ directory, create a deployment.yaml file for your application:

base/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
```

```
      image: my-app:1.0
      ports:
        - containerPort: 80
```

Next, create a kustomization.yaml file in the base directory to define the resources:

```
# base/kustomization.yaml

resources:

  - deployment.yaml
```

This kustomization.yaml tells Kustomize to manage the deployment.yaml file.

Step 2: Create Overlay for Development (dev)

In the overlays/dev/ directory, create a kustomization.yaml file:

overlays/dev/kustomization.yaml

```
resources:

  - ../../base

patchesStrategicMerge:

  - deployment.yaml
```

Here, the resources field includes the base configuration (../../base), and we also provide a patch for the deployment.yaml file to make environment-specific changes. Now, create the deployment.yaml patch for development:

```
# overlays/dev/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 2  # Increase replicas for the dev environment
  template:
    spec:
      containers:
        - name: my-app
          image: my-app:dev  # Use the dev version of the image
```

Step 3: Create Overlay for Production (prod)

In the overlays/prod/ directory, create a kustomization.yaml file:

```
# overlays/prod/kustomization.yaml
resources:
  - ../../base
patchesStrategicMerge:
  - deployment.yaml
```

And the deployment.yaml patch for production:

```
# overlays/prod/deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 5  # Increase replicas for the prod environment
  template:
    spec:
      containers:
        - name: my-app
          image: my-app:latest  # Use the latest production image
```

5. Apply Kustomized Resources

Once you've created the base and overlays, you can deploy your resources to a Kubernetes cluster.

For the Development Environment:

To apply the resources for the dev environment, navigate to the overlays/dev directory and run:

kubectl apply -k overlays/dev/

This will combine the base and the dev overlay and apply the configuration to your cluster. The resulting deployment will have 2 replicas and use the my-app:dev image.

For the Production Environment:

Similarly, for production:

kubectl apply -k overlays/prod/

This will apply the production-specific customizations, such as 5 replicas and using the my-app:latest image.

6. Advantages of Using Kustomize

1. No Templating Logic: Unlike Helm, Kustomize doesn't require complex templating, which means it's easier to use for simple customization.
2. Environment-Specific Configuration: You can easily apply environment-specific customizations (e.g., different numbers of replicas, different images) without modifying the base files.
3. Reusability: Kustomize encourages reusable configurations. You can define a base configuration that can be used in different environments, reducing duplication.
4. Declarative and Version-Control Friendly: Since Kustomize works with plain YAML files, it's easier to version control, and changes are clear and declarative.

7. Kustomize vs Helm

Kustomize:	**Helm:**
Primarily used for customizing existing YAML files (no templating).	Uses templating to generate Kubernetes YAML files from templates.
Works by layering and patching base configurations for different environments.	Suitable for more complex, reusable packages with variables, values files, and dependencies.
Ideal for simpler scenarios where you don't need the complexity of Helm charts.	Provides additional functionality like dependency management and versioning.

8. Conclusion

Kustomize is a powerful tool for Kubernetes developers, especially for CKADs who need to manage Kubernetes resources in a clean, environment-specific manner. By allowing you to customize resource definitions across multiple environments (like dev, staging, and prod) without modifying the original YAML files, Kustomize is a great option for managing Kubernetes resources in a maintainable way. With Kustomize, you can:

1. Create a base configuration and layer environment-specific customizations on top.
2. Easily apply customizations without modifying the base files.
3. Use simple declarative YAML files, which makes it version control-friendly and easier to understand.

By mastering Kustomize, you can streamline your Kubernetes deployment process and manage configurations more effectively, making it an essential tool in your toolkit as a Kubernetes Application Developer (CKAD).

Domain 4.1: Discover and Use Resources That Extend Kubernetes (CRDs, Operators)
As a Kubernetes Application Developer (CKAD), you will often need to extend the native capabilities of Kubernetes to meet the specific needs of your application or workload. This is where Custom Resources (CRDs) and Operators come into play. These resources allow you to customize and automate application management, creating a richer, more flexible Kubernetes ecosystem. In this section, we'll explore how to discover and use Custom Resource Definitions (CRDs) and Operators, and how they help extend Kubernetes functionality.

1. What Are Custom Resources (CRDs)?

- A Custom Resource (CR) is an extension of the Kubernetes API that allows you to add your own API objects, outside of the standard built-in Kubernetes resources (such as Pods, Deployments, and Services).
- Custom Resource Definitions (CRDs): A CRD is a way to define your own custom resources in Kubernetes. Once a CRD is created, you can manage instances of the custom resource just like any other Kubernetes object (e.g., Pods, Services, etc.) using kubectl.

Example use case: You might create a custom resource for a database configuration (e.g., DatabaseConfig) or for an application-specific resource like a CacheCluster.

2. How to Discover and Use CRDs

a) Checking for Available CRDs in Your Cluster
To discover available CRDs in your cluster, you can use the following kubectl command:

```
kubectl get crds
```

This will list all the custom resource definitions that are available in the cluster.
b) Creating a Custom Resource Definition (CRD) To extend Kubernetes with a new custom resource, you first need to define the CRD. A CRD is simply a YAML file that specifies the kind, metadata, and specifications of the custom resource.
Here is an example of a CRD definition for a custom resource called CacheCluster:

```
apiVersion: apiextensions.k8s.io/v1
kind: CustomResourceDefinition
metadata:
  name: cacheclusters.example.com
spec:
  group: example.com
  names:
    kind: CacheCluster
    listKind: CacheClusterList
    plural: cacheclusters
```

```
    singular: cachecluster
  scope: Namespaced
  versions:
  - name: v1
    served: true
    storage: true
    schema:
      openAPIV3Schema:
        type: object
        properties:
          spec:
            type: object
            properties:
              size:
                type: integer
                description: "Size of the cache"
                example: 3
              engine:
                type: string
                description: "Cache engine (e.g., Redis)"
                example: "redis"
```

- Kind: Defines the custom resource's name (CacheCluster).
- Scope: Defines whether the resource is namespaced or cluster-scoped. In this case, it's namespaced.
- Spec: Specifies the fields the resource will have, such as size and engine in the example.

Once the CRD is defined, you apply it to your Kubernetes cluster:

```
kubectl apply -f cachecluster-crd.yaml
```

After the CRD is applied, Kubernetes will start recognizing the custom resource.
c) Creating and Managing Custom Resources
Once the CRD is applied, you can create instances of your custom resource (in this case, CacheCluster):

Save this as my-cache-cluster.yaml and apply it to your cluster:

```
kubectl apply -f my-cache-cluster.yaml
```

You can now manage this resource like any other Kubernetes resource:
Get the resource:

```
kubectl get cacheclusters
```

Describe the resource:
kubectl describe cachecluster my-cache-cluster

Delete the resource:
kubectl delete cachecluster my-cache-cluster

3. What Are Kubernetes Operators?

An Operator is a method of packaging, deploying, and managing a Kubernetes application. It uses Custom Resources (CRs) to extend the Kubernetes API and make the application management more automated, handling lifecycle events such as installation, scaling, backups, and upgrades.
Operators are typically written as controllers (which watch for changes to resources and take actions accordingly). They are very useful for managing stateful applications that have complex lifecycle management needs.

4. How Operators Work with CRDs

Operators use CRDs to define the resources they manage. The operator watches for changes to these CRDs, and when a change occurs, it takes actions to reconcile the desired state of the resource with the actual state. For example, if you have a CacheCluster custom resource, an operator would ensure that the right number of Redis instances are running and automatically scale them up or down based on the size field in the CacheCluster spec.

5. How to Discover, Install, and Use Operators

a) Discovering Available Operators
You can discover available Operators through OperatorHub, a catalog of Kubernetes Operators. To browse OperatorHub, visit the OperatorHub.io website. Additionally, many Kubernetes distributions like OpenShift or Rancher have integrated operator catalogs.
b) Installing Operators
Operators are typically installed as Deployments or StatefulSets within a cluster. You can install Operators either manually by applying a YAML file or through Helm. Example Installation via YAML: Suppose you want to install a Redis operator, which can manage Redis clusters using the RedisCluster custom resource. First, find the Operator's installation YAML (often provided in the official documentation of the Operator).
You can install the Redis operator with:
kubectl apply -f redis-operator.yaml

Example Installation via Helm: If the operator is available as a Helm chart, you can install it using Helm:
helm install redis-operator stable/redis-operator

c) Using Operators
Once the operator is installed, you can create instances of the custom resource it manages. For example, if you're using a Redis operator, you might define a RedisCluster custom resource to create a Redis cluster:

```
apiVersion: redis.k8s.io/v1
kind: RedisCluster
metadata:
  name: my-redis-cluster
spec:
  size: 3
  redisVersion: "6.0.5"
```

Apply it using:

kubectl apply -f my-redis-cluster.yaml

The operator will then create the necessary Redis Pods and manage the cluster lifecycle automatically.
d) Updating Operators and Custom Resources

- Updating the Operator: To update the operator, you would update the Deployment or StatefulSet running the operator, either through a new version of the YAML or a Helm chart.
- Updating the Custom Resource: To update a custom resource (e.g., RedisCluster), modify the resource definition and apply the changes using kubectl apply.

6. Benefits of CRDs and Operators

1. Automated Management: Operators enable the automation of complex application lifecycle management tasks such as installation, configuration, backup, scaling, and upgrades.
2. Custom Resource Management: CRDs allow you to define custom resources that meet the unique needs of your application.
3. Seamless Integration with Kubernetes: Both CRDs and Operators are first-class citizens in the Kubernetes ecosystem. They integrate seamlessly into Kubernetes' native management tools (kubectl, Helm, etc.).
4. Declarative Model: Both CRDs and Operators use the declarative model of Kubernetes, which ensures that the desired state of the application is continuously maintained by the Kubernetes controller.

7. Conclusion

In this section, we have covered how to discover and use resources that extend Kubernetes, specifically focusing on Custom Resource Definitions (CRDs) and Operators.

- CRDs allow you to create custom resources that extend Kubernetes' capabilities and define your own objects for managing applications.
- Operators provide an automated way to manage the lifecycle of complex applications by leveraging CRDs to handle the installation, configuration, scaling, and maintenance of stateful applications.

Mastering these concepts will significantly enhance your ability to work with Kubernetes, enabling you to build more flexible, scalable, and automated systems that go beyond the built-in Kubernetes resources.

Domain 1.4: Utilize Persistent and Ephemeral Volumes

In Kubernetes, volumes are essential for providing storage to containers within Pods. Containers are typically ephemeral, meaning that when a container is terminated or rescheduled, any data stored in its filesystem is lost. To manage data more effectively, Kubernetes offers two primary types of volumes:

- Ephemeral and
- Persistent.

As a Kubernetes Application Developer (CKAD), understanding when and how to use these different types of volumes is crucial for building scalable and resilient applications. This section will explain the difference between ephemeral and persistent volumes, provide use cases, and show how to configure them.

1. Ephemeral Volumes

- Ephemeral volumes are temporary storage that exists for the duration of a Pod.
- Once the Pod is deleted, the data stored in these volumes is also deleted.
- Ephemeral volumes are ideal for scenarios where data persistence is not required or where data can be re-created easily (e.g., temporary files, caches, or scratch space).

Common Ephemeral Volume Types:

1. emptyDir: A simple empty directory that is created when a Pod is scheduled on a node and exists as long as the Pod is running. You cannot directly list an "emptyDir" volume in Kubernetes because it is a temporary, ephemeral volume created on the node where the pod is running and is not managed as a separate entity that can be listed independently; you can only see its existence within the context of a pod's manifest when defining it as a volume type.

Key points about emptyDir:

- Created on Pod assignment: An emptyDir volume is created when a pod is assigned to a node.
- Transient data: Data stored in an emptyDir is deleted when the pod is removed from the node, even if the container restarts within the pod.
- Accessing in Pod: To see the emptyDir volume, you would need to access the pod itself and list the files within the mounted path specified in the pod manifest.

2. configMap: Stores configuration data that can be injected into Pods as files.
3. secret: Stores sensitive information, such as passwords or tokens, which are injected into Pods.
4. downwardAPI: Provides metadata about the Pod, such as labels, annotations, or resource usage metrics.
5. projected: A volume that aggregates multiple sources into one volume, such as a mix of secrets, config maps, or downward API data.

Use Cases for Ephemeral Volumes:

1. Temporary files: Storing files that don't need to persist beyond the life of a Pod (e.g., logs, caches).
2. Configuration data: Using configMap or secret volumes to inject configuration files or environment variables into Pods.
3. Shared memory: Using emptyDir for sharing files between containers within the same Pod (e.g., a web server and a sidecar container).

Example: emptyDir Volume

Here's an example of using an emptyDir volume in a Pod to store temporary data:

```
apiVersion: v1
kind: Pod
metadata:
  name: ephemeral-volume-example
spec:
  containers:
    - name: main-container
      image: busybox
      command: ["sh", "-c", "echo 'Hello World' > /data/hello.txt; sleep 3600"]
      volumeMounts:
        - mountPath: /data
          name: emptydir-volume
  volumes:
    - name: emptydir-volume
      emptyDir: {}
```

In this example:

- The emptyDir volume is created for the Pod, and the main container writes data to the /data directory.
- The data stored in emptyDir will only exist for as long as the Pod is running. If the Pod is deleted, the data is lost.

ConfigMaps

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: log-config
data:
  username: k8s-admin
  access_level: "1"
---
apiVersion: v1
kind: Pod
metadata:
  name: configmap-pod
spec:
  containers:
    - name: test
      image: busybox:1.28
      command: ['sh', '-c', 'echo "The app is running!" && tail -f /dev/null']
      volumeMounts:
        - name: config-vol
```

```
        mountPath: /etc/config
  volumes:
    - name: config-vol
      configMap:
        name: log-config
        #items:
        #  - key: username
        #    path: keys
         # - key: access_level
         #   path: keys
#https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/
```

Secrets

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
data:
  username: bXktYXBw
  password: Mzk1MjgkdmRnN0pi
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: nginx
      volumeMounts:
        # name must match the volume name below
        - name: secret-volume
          mountPath: /etc/secret-volume
          readOnly: true
  # The secret data is exposed to Containers in the Pod through a Volume.
  volumes:
    - name: secret-volume
      secret:
        secretName: test-secret
#https://kubernetes.io/docs/tasks/inject-data-application/distribute-credentials-secure/#set-posix-permissions-for-secret-keys
```

2. Persistent Volumes (PVs)

- Unlike ephemeral volumes, persistent volumes are designed to store data that needs to persist beyond the life of a Pod. This makes them ideal for applications that require durable storage, such as databases or file storage.
- Lifecycle independent of Pods: PVs exist independently of Pods and retain data even if a Pod is deleted or rescheduled.
- Persistent Volumes can be backed by various storage systems such as:
    - local disks,
    - hostPath
    - cloud-based block storage (e.g., AWS EBS, GCE Persistent Disk),
    - or network-attached storage (e.g., NFS, GlusterFS, CEPH, iSCSI).
- Access Modes: PVs can have different access modes, determining how they can be mounted by Pods. These include:
    - ReadWriteOnce (RWO): The volume can be mounted as read-write by a single node.
    - ReadOnlyMany (ROX): The volume can be mounted as read-only by many nodes.
    - ReadWriteMany (RWX): The volume can be mounted as read-write by many nodes.

Use Cases for Persistent Volumes:

1. Databases: Storing the data of stateful applications, such as databases, that require persistent storage.
2. Shared storage: Enabling multiple Pods to access the same storage for read/write purposes (e.g., file sharing).
3. Long-term data storage: For applications that need to store user data, logs, or backups.

Key Concepts for Persistent Volumes:

1. PersistentVolume (PV): A resource in the cluster that represents a piece of storage. It's provisioned by an administrator or dynamically provisioned using StorageClasses.
2. PersistentVolumeClaim (PVC): A request for storage by a user/application. A PVC is bound to a PV that meets its storage requirements.
3. StorageClass: Defines the type of storage to be used (e.g., standard, SSD, NFS) and allows for dynamic provisioning of PVs.

3. How PVs, PVCs, and Storage Classes Work Together

PV and PVC:

When an application requires storage, a PersistentVolumeClaim (PVC) is created by the user or application, specifying the desired storage size, access mode, and storage class (optional). Binding a PVC to a PV: Kubernetes searches for an available Persistent Volume (PV) that meets the requirements specified in the PVC. If a matching PV exists, Kubernetes binds the PVC to the PV. If no suitable PV exists and the PVC has a storageClassName, Kubernetes may trigger the dynamic provisioning of a new PV based on the specified

StorageClass.

A StorageClass is a Kubernetes resource that defines the type of storage to be used for dynamic provisioning. Storage Classes specify which storage backends can be used (e.g., AWS EBS, GCP Persistent Disks, Ceph, etc.), as well as parameters such as IOPS (Input/Output Operations Per Second), replication, and encryption.

Dynamic Provisioning:

When dynamic provisioning is enabled (via a StorageClass), the provisioner defined in the StorageClass will automatically provision a PV that meets the criteria specified in the PVC. For example, if you use AWS as the provisioner, a new EBS volume will be created and bound to the PVC. Releasing and Reclaiming a PV: Once the PVC is no longer needed (e.g., the application is deleted), the PVC is deleted, and the bound PV becomes available. The reclaim policy defined in the PV (either Retain, Delete, or Recycle) determines what happens to the data on the PV and whether the storage resource is deleted or retained for future use.

Key Features of Storage Classes:

- Dynamic Provisioning: When a PVC specifies a storageClassName, Kubernetes will use the associated Storage Class to provision a PV automatically if no matching PV exists.
- Provisioning Parameters: You can define parameters for different storage backends (e.g., for AWS, GCP, etc.).
- Reclaim Policy: The StorageClass can define the reclaim policy for PVs (whether to delete the volume, retain it, or recycle it).

Example Storage Class YAML:

Here's an example of a Storage Class definition:

```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: standard
provisioner: kubernetes.io/aws-ebs
parameters:
  type: gp2
  iopsPerGB: "10"
reclaimPolicy: Retain
```

This configuration:

- Provisioner: The storage will be provisioned by the kubernetes.io/aws-ebs provisioner.
- Parameters: Defines specific parameters for the AWS EBS volumes, such as the volume type (gp2) and IOPS per GB.
- Reclaim Policy: The Retain policy will ensure that the volume is not deleted when the PVC is deleted.

4. EXAMPLE Walkthrough of PV, PVC and StorageClasses

Persistent Volume

Here's an example that demonstrates how to use a PersistentVolume (PV) and PersistentVolumeClaim (PVC):

PersistentVolume (PV) definition:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 1Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  #storageClassName: standard
  hostPath:
    path: /mnt/data
```

This configuration:

- Storage: 1Gi of storage.
- Access Mode: Mounted as read-write by a single node (ReadWriteOnce).
- Reclaim Policy: The PV is retained after it is released from a PVC (Retain), meaning the data remains available.
- Storage Class: The standard class will be used for dynamic provisioning.

PersistentVolumeClaim (PVC) definition:

- Claims resources from available PVs: PVCs match and bind to available PVs that satisfy the requested storage capacity and access modes.
- Dynamic provisioning: When a PVC is created, Kubernetes can dynamically provision a PV if there is no available one that matches the PVC's requirements. This is managed through Storage Classes.
- Storage Class Awareness: PVCs can specify a storageClassName, which determines how the underlying PV is provisioned (either dynamically or statically).

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

This configuration:

- Storage: 1Gi of storage.
- Access Mode: Mounted as read-write by a single node (ReadWriteOnce).
- Reclaim Policy: The PV is retained after it is released from a PVC (Retain), meaning the data remains available.
- Storage Class: The standard class will be used for dynamic provisioning.
- The PVC specifies a storage request of 1Gi and is bound to a PV that can fulfill these requirements.
- The accessModes and storageClassName should match the PV's definition.

Pod Using PVC:

```
apiVersion: v1
kind: Pod
metadata:
  name: pvc-pod
spec:
  containers:
    - name: main-container
      image: busybox
      command: ["sh", "-c", "echo 'Persistent storage' > /mnt/data/storage.txt; sleep 3600"]
      volumeMounts:
        - mountPath: /mnt/data
          name: pvc-storage
  volumes:
    - name: pvc-storage
      persistentVolumeClaim:
        claimName: my-pvc
```

In this setup:

1. The Pod uses a PersistentVolumeClaim (PVC) to request storage from the PV.
2. The container writes data to /mnt/data/storage.txt, which is backed by the persistent volume.

Reclaim Policies

The Reclaim Policy is an important setting in both PVs and Storage Classes that dictates what happens to the persistent volume when the associated PVC is deleted.

- Retain: When the PVC is deleted, the PV will not be deleted. The data remains on the disk, and the PV will go into the Released state. It must be manually cleaned and reused or deleted.
- nDelete: When the PVC is deleted, the PV is also deleted, and the underlying storage is cleaned up (for dynamic provisioning).
- Recycle: The PV is scrubbed (data is wiped) and made available for reuse (this reclaim policy is deprecated and not widely used).

Example of Reclaim Policy:

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: my-pv
spec:
  capacity:
    storage: 10Gi
  persistentVolumeReclaimPolicy: Retain
```

Access Modes

The accessMode defines how a PV can be accessed by Pods:

- ReadWriteOnce (RWO): The volume can be mounted as read-write by a single node.
- ReadOnlyMany (ROX): The volume can be mounted as read-only by many nodes.
- ReadWriteMany (RWX): The volume can be mounted as read-write by many nodes.

Example Access Mode:

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 5Gi
```

Use Case Examples

a. Shared Filesystem with ReadWriteMany You can use a ReadWriteMany (RWX) access mode if you need to mount a volume that allows multiple nodes to read/write to the volume, useful for shared storage scenarios like file systems (e.g., NFS). b. Cloud Volumes for Stateful Apps If you're running a stateful application (like a database) in your Kubernetes cluster, you might use a ReadWriteOnce (RWO) volume, such as an AWS EBS volume or GCP Persistent Disk, which is tied to a single node but ensures data persistence.

Difference Between Ephemeral and Persistent Volumes

Feature	Ephemeral Volumes	Persistent Volumes
Lifetime	Tied to the lifecycle of a Pod.	Exists independently of Pods and can persist beyond Pod lifecycle.

Use Case	Temporary storage for caches, logs, or scratch data.	Durable storage for stateful applications like databases.
Data Loss	Data is lost when the Pod is deleted.	Data persists even if the Pod is deleted.
Backends	Typically local (e.g., emptyDir, configMap, secret).	Can be backed by cloud storage, NFS, or other persistent storage solutions.
Access Modes	Read/write for a single Pod/container.	Access modes like ReadWriteOnce, ReadOnlyMany, or ReadWriteMany for multiple Pods.

5. Best Practices for Using Volumes

- Use Ephemeral Volumes for Temporary Data: If your application generates temporary data that doesn't need to persist after the Pod terminates, use ephemeral volumes (e.g., emptyDir, configMap, or secret).
- Use Persistent Volumes for Stateful Applications: For applications that require data persistence (e.g., databases, logging systems), use persistent volumes backed by cloud storage or networked file systems.
- Claim Resources Efficiently: When using Persistent Volumes, ensure that your PVCs request appropriate storage resources based on your application's needs, and be mindful of storage classes and capacity.
- Security Considerations: When using sensitive data, consider using secret volumes for injecting secrets (e.g., passwords, API keys) into Pods securely.

6. Conclusion

As a Kubernetes Application Developer (CKAD), understanding how to use both ephemeral and persistent volumes is critical for building applications that are both scalable and resilient. By using ephemeral volumes for temporary data and persistent volumes for data that needs to be stored long-term, you can ensure your applications are both performant and reliable. Understanding these concepts and how they interconnect is critical for successfully managing storage in Kubernetes and ensuring that your applications have the persistent storage they need, while maintaining flexibility and scalability.

- Persistent Volumes (PVs) provide a way to abstract storage resources in the cluster, making it easy to manage storage.
- Persistent Volume Claims (PVCs) allow users to request storage without worrying about how it's provisioned.
- Storage Classes define how storage should be dynamically provisioned, with specific parameters based on your environment (e.g., cloud provider).

Make sure to:

- Use ephemeral volumes for temporary data that doesn't need to survive beyond the Pod.

- Use persistent volumes for stateful applications or when you need data persistence across Pod restarts or node failures.
- Proper volume management is a key component of Kubernetes application development, and mastering it will help you build efficient and effective cloud-native applications.

Domain 5.2: Provide and Troubleshoot Access to Applications via Services

In Kubernetes, Services are an abstraction that defines how to expose and access Pods in a network. Services provide a stable IP address or DNS name for accessing Pods, abstracting the complexity of dynamically changing Pod IP addresses as Pods are created, destroyed, or rescheduled across nodes.
In this section, we'll cover:

- How to create and configure Services in Kubernetes.
- How to troubleshoot Service access issues.
- Common Service types and their use cases.
- Key concepts and tools used in diagnosing Service-related problems.

1. Overview of Kubernetes Services

Kubernetes Services expose Pods to network traffic and are essential for providing reliable, stable access to applications. By default, Pods are ephemeral (i.e., they can be created and destroyed dynamically), so Services provide an abstraction that helps ensure communication with Pods, even if their IPs change.

Types of Services:

There are four main types of Kubernetes Services:

- ClusterIP (default): Exposes the service on a cluster-internal IP. This makes the Service accessible only from within the cluster.
- NodePort: Exposes the service on a static port on each node's IP. This allows access to the Service from outside the cluster.
- LoadBalancer: Provisions a load balancer (if supported by the cloud provider) to expose the Service externally.
- ExternalName: Maps the service to an external DNS name, allowing Kubernetes resources to access external services using a DNS alias.

Key Concepts:

- ClusterIP is the most common type and is used when the service does not need to be accessed externally.
- Endpoints are the set of Pods that are selected by a Service's label selector.

2. Creating a Kubernetes Service

Here's an example of how to create a Service in Kubernetes. Service Example: Exposing an Application Using ClusterIP This YAML file creates a Service of type ClusterIP that exposes a Pod running an HTTP server.

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-service
spec:
  selector:
    app: my-app
  ports:
```

```
  - protocol: TCP
    port: 80
    targetPort: 8080
clusterIP: None
```

Explanation:

- selector: Selects the Pods labeled app=my-app to associate with the Service.
- ports: The Service exposes port 80 and forwards traffic to Pods on port 8080.
- clusterIP: None: Specifies that the Service does not have a specific ClusterIP, which makes it headless. (This is useful when using StatefulSets.)
- Service Example: Exposing an Application Using NodePort

A NodePort service exposes your application outside the Kubernetes cluster. Below is an example of a NodePort service:

```
apiVersion: v1
kind: Service
metadata:
  name: my-app-nodeport
spec:
  selector:
    app: my-app
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
      nodePort: 30007
  type: NodePort
```

Explanation:

- nodePort: 30007: The service will be accessible externally on port 30007 on every node in the cluster.
- type: NodePort: This type exposes the service on a static port across all nodes in the cluster.

3. Troubleshooting Service Access

Sometimes, access to services in Kubernetes may not work as expected. There are several common reasons for Service-related issues. Here are steps to troubleshoot these problems:

Common Issues:

Service not accessible externally: The most common reason for this is that the service type is incorrectly set or the NodePort or LoadBalancer is not properly configured.

- Pods not selected by the Service: The Service's label selector may not match the Pods you want to expose.
- Service not routing traffic to Pods: Misconfigured targetPort or issues with the underlying Endpoints can result in traffic not reaching the Pods.
- DNS issues: If the application cannot be accessed using the Service name, there may be issues with DNS resolution inside the cluster.

Troubleshooting Steps:

1. Check the Service Configuration Verify that the Service is created and configured correctly.

kubectl get svc my-app-service

Check if the Service is of the correct type (ClusterIP, NodePort, LoadBalancer) and if the correct ports are exposed.

2. Check Service Endpoints A Service relies on Endpoints to route traffic to Pods. If no Endpoints are associated with the Service, traffic won't reach the Pods.

kubectl get endpoints my-app-service

If no Endpoints are listed, check if the Pod label selector is correct and whether Pods matching the selector are running.

3. Verify Pod Labels and Selectors Ensure that the labels of the Pods match the Service's label selector.

kubectl get pods --show-labels

For the Service to work correctly, the Pods should have labels that match the selector in the Service definition.

4. Check Pod Logs If the Pods are selected correctly but the service still doesn't respond, check the Pod logs to ensure the application inside the Pods is functioning correctly.

kubectl logs <pod-name>

Check if there are any issues with the application that might be preventing it from accepting traffic on the specified port.

5. Verify Network Connectivity You can use the kubectl exec command to troubleshoot network connectivity to the Pods from other Pods.

kubectl exec -it <pod-name> -- curl <service-name>:<service-port>

This command tests whether a Pod can reach the Service and port. This is useful for diagnosing network issues between Pods.

6. Investigate NodePort and LoadBalancer Services If using a NodePort or LoadBalancer service, check that the service is exposed properly. NodePort: Ensure that the port is accessible on all the nodes in the cluster.

kubectl get svc my-app-nodeport

LoadBalancer: If using a LoadBalancer service, verify if the cloud provider has provisioned the external load balancer and that the IP is accessible.
kubectl describe svc my-app-lb

Check if the EXTERNAL-IP is listed and whether it's reachable from outside the cluster.

7. DNS Resolution If the application is not reachable by its Service name, ensure that the DNS service in Kubernetes is running correctly. You can check if DNS resolution is working properly by querying the DNS from within a Pod:

kubectl exec -it <pod-name> -- nslookup my-app-service

4. Best Practices for Exposing Applications via Services

1. Use the Correct Service Type:
- Use ClusterIP for internal applications.
- Use NodePort or LoadBalancer for external access.
- ExternalName is useful for redirecting traffic to external services via DNS.
2. Match Labels Correctly: Ensure that the label selectors on the Service match the labels on the Pods to ensure correct routing of traffic.
3. Health Checks and Probes: Always implement readiness and liveness probes in your Pods. These help ensure that Pods are ready to handle traffic before the Service routes requests to them.
4. Limit Port Exposure: Only expose the necessary ports for each application. This minimizes the attack surface and limits the potential for unauthorized access.
5. Use Namespace Isolation: If your application spans multiple namespaces, ensure that the appropriate namespace selectors are used in the Service definition.

5. Conclusion

Providing and troubleshooting access to applications in Kubernetes using Services is a fundamental part of managing Kubernetes environments. Services provide a stable interface to access Pods, abstracting away the complexities of pod IP address management. By understanding the types of services available and troubleshooting common issues, you can ensure that your applications are reliably exposed to the network and that access problems can be quickly diagnosed and fixed.

Key takeaways:

- Services expose Pods and enable stable access to applications.
- Use the correct Service type based on your use case (ClusterIP, NodePort, LoadBalancer, ExternalName).
- Troubleshoot by verifying Service configurations, labels, endpoints, and Pod health.
- Leverage DNS and network debugging tools to identify connectivity issues.

Domain 5.3: Use Ingress Rules to Expose Applications

In Kubernetes, Ingress is an API object that manages external access to services within a cluster, typically HTTP and HTTPS traffic. Ingress allows you to expose HTTP/S-based applications in a Kubernetes cluster to the outside world, providing advanced routing, SSL termination, load balancing, and host-based or path-based routing.
In this section, we will cover:

- How to define and configure Ingress rules.
- Common use cases for Ingress in Kubernetes.
- How to troubleshoot Ingress-related issues.
- Best practices when using Ingress to expose applications.

1. Overview of Kubernetes Ingress

Ingress is a powerful Kubernetes resource that allows you to configure external HTTP/S access to your services. Unlike Services, which provide simple access to a group of Pods within the cluster, Ingress allows you to define fine-grained routing rules and handle SSL termination, path-based routing, and more.

Components of Ingress:

- Ingress Resource: The configuration for how external HTTP/S traffic should be routed to Services inside the cluster.
- The Ingress is a Kubernetes resource that lets you configure an HTTP load balancer for applications running on Kubernetes, represented by one or more Services. Such a load balancer is necessary to deliver those applications to clients outside of the Kubernetes cluster.
- The Ingress resource supports the following features:
    - Content-based routing:
    - Host-based routing. For example, routing requests with the host header foo.example.com to one group of services and the host header bar.example.com to another group.
    - Path-based routing.
      For example, routing requests with the URI that starts with /serviceA to service A and requests with the URI that starts with /serviceB to service B. TLS/SSL termination for each hostname, such as foo.example.com. See the Ingress User Guide to learn more about the Ingress resource.
- Ingress Controller: A controller that implements the rules defined in the Ingress resource. It is responsible for managing and directing traffic based on Ingress rules. Popular controllers include NGINX, Traefik, and HAProxy.
- The Ingress Controller is an application that runs in a cluster and configures an HTTP load balancer according to Ingress resources. The load balancer can be a software load balancer running in the cluster or a hardware or cloud load balancer running externally. Different load balancers require different Ingress Controller implementations.
- In the case of NGINX, the Ingress Controller is deployed in a pod along with the load balancer.

Benefits of Using Ingress:

- URL-based Routing: Route traffic to different services based on the URL path (e.g., /app1 to one service, /app2 to another).
- Host-based Routing: Route traffic to services based on the host domain (e.g., example.com vs app.example.com).
- SSL Termination: Terminate SSL/TLS at the Ingress controller, offloading the burden from individual services.
- Path-based Routing: Route traffic to different services based on request paths.

2. How to Define Ingress Rules

Ingress rules are defined using a Kubernetes Ingress resource. A basic Ingress configuration looks like this:
Example: Basic Ingress Resource

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

Explanation:

- host: The domain name (my-app.example.com) that will be used to route traffic to this Ingress.
- http: This section defines HTTP routing rules.
- paths: Defines URL paths and the backend service to forward traffic to. In this case, any traffic to / will be directed to my-app-service on port 80.
- pathType: Defines how the path should be interpreted (Prefix matches the path and everything that starts with it, Exact matches only the exact path).

Example: Ingress with Path and Host-based Routing

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: multi-path-ingress
 namespace: default
spec:
 rules:
  - host: example.com
    http:
     paths:
      - path: /app1
        pathType: Prefix
        backend:
```

```
          service:
            name: app1-service
            port:
              number: 80
      - path: /app2
        pathType: Prefix
        backend:
          service:
            name: app2-service
            port:
              number: 80
```

Explanation:

- Traffic to example.com/app1 will be routed to app1-service, while traffic to example.com/app2 will be routed to app2-service.

Example: Ingress with SSL Termination

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: my-ssl-ingress
  annotations:
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
spec:
  rules:
    - host: secure-app.example.com
      http:
        paths:
          - path: /
            pathType: Prefix
            backend:
              service:
                name: secure-app-service
                port:
                  number: 443
  tls:
    - hosts:
        - secure-app.example.com
      secretName: my-ssl-secret
```

Explanation:

- This configuration enables SSL termination at the Ingress controller for the host secure-app.example.com.
- The SSL certificate is stored in a Secret (my-ssl-secret), which is used by the Ingress controller to handle HTTPS traffic.
- The nginx.ingress.kubernetes.io/ssl-redirect annotation ensures HTTP requests are redirected to HTTPS.

3. Setting Up an Ingress Controller

In order to use Ingress, you need to have an Ingress controller running in your cluster. Kubernetes does not come with a built-in Ingress controller, so you will need to deploy one. Here's an example of how to deploy the NGINX Ingress Controller:
Deploy NGINX Ingress Controller Create the NGINX Ingress Controller using Helm or kubectl. To install NGINX via Helm, run:
helm install nginx-ingress ingress-nginx/ingress-nginx --namespace kube-system

Alternatively, to deploy using kubectl:

kubectl apply -f
https://raw.githubusercontent.com/kubernetes/ingress-nginx/main/deploy/static/provider/cloud/deploy.yaml

This will create all necessary resources for NGINX to act as your Ingress controller.
Once deployed, verify that the controller is running:

kubectl get pods -n kube-system

You can check if the controller has an external IP address by running:

kubectl get svc -n kube-system

Lab Walkthrough:

1. Install the Ingress controller using helm

helm upgrade --install ingress-nginx ingress-nginx --repo https://kubernetes.github.io/ingress-nginx --namespace ingress-nginx --create-namespace

Expected output:
The ingress-nginx controller has been installed.

It may take a few minutes for the load balancer IP to be available.

You can watch the status by running 'kubectl get service --namespace ingress-nginx ingress-nginx-controller --output wide --watch'

An example Ingress that makes use of the controller:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: example
  namespace: foo
spec:
  ingressClassName: nginx
  rules:
    - host: www.example.com
      http:
        paths:
          - pathType: Prefix
            backend:
              service:
                name: exampleService
                port:
                  number: 80
            path: /
  # This section is only required if TLS is to be enabled for the Ingress
  tls:
    - hosts:
      - www.example.com
      secretName: example-tls
```

If TLS is enabled for the Ingress, a Secret containing the certificate and key must also be provided:

```
apiVersion: v1
kind: Secret
metadata:
  name: example-tls
  namespace: foo
data:
  tls.crt: <base64 encoded cert>
  tls.key: <base64 encoded key>
type: kubernetes.io/tls
```

2. Check that ingresscontroller is running

kubectl get pods --namespace ingress-nginx

Expected output:

```
NAME                                        READY   STATUS    RESTARTS   AGE
ingress-nginx-controller-5f649bbff8-gs8j2   1/1     Running   0          89s
```

3. Check the Ingress Controller and take note of the EXTERNAL-IP. You must see 'localhost'. If you dont see localhost, check and delete any service of type 'LoadBalancer'.

```
kubectl get service ingress-nginx-controller --namespace=ingress-nginx
```

Expected output:

```
NAME                       TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)                      AGE

ingress-nginx-controller   LoadBalancer   10.107.184.189   localhost     80:31210/TCP,443:30943/TCP   4m49s
```

4. Create deployment and service objects. Put the information in deploy1.yaml

```
 apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx
spec:
 replicas: 1
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
     - name: nginx
       image: nginx
       ports:
       - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
 name: nginx
spec:
 type: NodePort
 selector:
   app: nginx
```

```
ports:
  - port: 80
    targetPort: 80
```

Expected output:
User "demo-user" set.

5. create the objects

kubectl apply -f deploy1.yaml

Expected output:

deployment.apps/nginx created

service/nginx created

6. Create ingress resource. Put the following in ingress.yaml

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

7. Create the ingress resource

kubectl apply -f ingress.yaml

Expected output.
ingress.networking.k8s.io/example-ingress created

8. check the deployment resources created

kubectl get pods

Expected output.

```
NAME                READY   STATUS    RESTARTS      AGE

nginx-7c5ddbdf54-7ccnq   1/1     Running   0          2m11s
```

9. Check the service object

kubectl get svc

Expected output. Your output may differ

```
nginx           NodePort    10.106.222.193   <none>        80:30175/TCP       58s
```

10. Test ingress. Put localhost:30175 on the browser or use curl as follows

curl localhost:30175

```
 <!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>
<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>
<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

11. Create another path by using Apache image

Create deployment and service objects. Put the information in deploy2.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: apache
spec:
 replicas: 1
 selector:
   matchLabels:
     app: apache
 template:
   metadata:
     labels:
       app: apache
   spec:
     containers:
     - name: apache
       image: httpd
       ports:
       - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
 name: apache
spec:
 type: NodePort
 selector:
   app: apache
 ports:
   - port: 80
     targetPort: 80
```

12. create the objects

kubectl apply -f deploy2.yaml

Expected output:
deployment.apps/apache created

service/apache created

13. Modify the ingress resource. Add to the ingress.yaml

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
 name: example-ingress
spec:
 rules:
 - host: nginx-hello.info
   http:
     paths:
     - pathType: Prefix
       path: "/"
       backend:
         service:
           name: nginx
           port:
             number: 80
   - pathType: Prefix
       path: "/apache"
       backend:
         service:
           name: apache
           port:
             number: 80
```

14. Create the ingress resource

kubectl apply -f ingress.yaml

Expected output.
ingress.networking.k8s.io/example-ingress configured

15. check the deployment resources created

kubectl get pods

Expected output.

```
NAME                     READY   STATUS    RESTARTS      AGE

apache-644bc8865b-7gz8c   1/1    Running   0             11s

nginx-7c5ddbdf54-7ccnq    1/1    Running   0             2m11s
```

16. Check the service object

```
kubectl get svc
```

Expected output. Your output may differ

```
nginx          NodePort    10.106.222.193   <none>       80:30175/TCP       58s

apache         NodePort    10.96.150.54     <none>       80:31125/TCP       4m
```

17. Test ingress. Put localhost:30175 on the browser or use curl as follows

```
curl localhost:30175
```

4. Troubleshooting Ingress Issues

If you're experiencing issues with your Ingress configuration, there are several key areas to check:

1. Verify Ingress Resource Configuration

Check if the Ingress resource is properly created and configured.

```code

kubectl get ingress my-app-ingress -o yaml

Ensure that the host, paths, and backend service names are correctly defined.

2. Check Ingress Controller Logs

The Ingress controller is responsible for routing the traffic based on your rules. If traffic is not being routed correctly, check the logs of your Ingress controller.
For NGINX Ingress, you can run:
kubectl logs <nginx-ingress-pod> -n kube-system

Look for errors related to configuration, missing services, or SSL issues.

3. Verify Service and Pod Availability Ensure that the backend Service and Pods are running and reachable by the Ingress controller.

kubectl get svc my-app-service

kubectl get pods -l app=my-app

Verify that the service exists and that there are Pods available to handle requests.

4. Check DNS Resolution Ensure that the DNS for the Ingress host is correctly set up. For example, if you're using my-app.example.com, make sure that the DNS record points to the Ingress controller's external IP address or LoadBalancer IP. Use nslookup or dig to verify:

nslookup my-app.example.com

5. SSL/TLS Issues If you're using SSL, verify that the SSL certificate is correctly installed and the TLS Secret is present.

kubectl get secret my-ssl-secret

If there are issues with SSL/TLS, check the Ingress controller logs for SSL-related errors.

6. Best Practices for Using Ingress
```

- Use Annotations for Fine-grained Control: Many Ingress controllers (like NGINX) support annotations for additional configuration, such as SSL redirection, rewrite-targets, rate limiting, etc. Make sure to utilize them when needed.
- Leverage Path and Host-Based Routing: Organize your services logically by exposing multiple applications through the same domain using path-based or host-based routing. This reduces the need for multiple public IPs or LoadBalancers.
- Secure Ingress with SSL: Always use SSL/TLS for secure communication. Use Kubernetes Secrets to manage SSL certificates and configure your Ingress controller to terminate SSL/TLS traffic.
- Monitor and Log Traffic: Enable logging and monitoring in the Ingress controller to troubleshoot issues and monitor traffic patterns.
- Consider Using a Centralized Ingress Resource: If you have multiple namespaces and applications, consider using a centralized Ingress resource or controller for consistent routing policies across all applications.

6. Conclusion

Ingress provides a powerful way to expose Kubernetes applications to the outside world. It offers advanced features like path-based routing, SSL termination, and host-based routing, which makes it ideal for complex applications with multiple services. By understanding how to define Ingress resources and troubleshoot issues, you can ensure that your Kubernetes applications are accessible and functioning as intended.

Key Takeaways:

- Ingress allows advanced HTTP/S routing for Kubernetes applications.
- Ingress controllers are needed to implement Ingress rules, with NGINX being a popular choice.
- Ensure DNS, SSL, and Service configurations are correctly set up.

Troubleshoot

- issues by checking logs, resources, and DNS configuration.
- Follow best practices for securing and organizing your Ingress resources.