Percipient - course Pods ….summary

1. Identify what a Pod is & Explain its crucial role in Kubernetes

What is a Pod?

A Pod is the smallest deployable and schedulable unit in Kubernetes.
It acts as a wrapper around:

- One container (most common case), or
- Multiple tightly coupled containers that must run together

Key characteristics

- **Shared execution environment** → containers inside a pod share:
  - Storage volumes
  - Network namespace (same IP & port space)
  - IPC (inter-process communication)
- Pods represent a **logical host** for an application.
- Containers inside a pod are always:
  - **Co-located** (run on the same node)
  - Co-scheduled
  - **Share resources** and are often complementary (e.g., Web server + sidecar)

Why Pods are crucial

- They define how an application runs in Kubernetes.
- They act as the **basic building block** upon which higher-level objects (Deployments, ReplicaSets, DaemonSets, Jobs) operate.
- They group containers that belong together and need to communicate efficiently.
- Each pod gets its **own unique IP address**, simplifying networking.
- Support init containers and ephemeral containers for:
  - Initialization sequences
  - Debugging

How Pods behave

- Pods are **ephemeral**: they don't self-heal.
- If a pod fails, Kubernetes **creates a new pod** rather than fixing the old one.
- Controlled by workload resources (Deployments, Jobs, etc.) using **PodTemplates**.

✅ 2. Identify the phases of a Pod's life & Describe its lifecycle

Pod Lifecycle Phases

Kubernetes tracks pod progress through the following **Pod phases**:

1. Pending
   - Pod is accepted by the cluster but containers haven't started yet.
   - Includes time for scheduling + image downloads.
2. Running
   - Pod has been scheduled to a node.
   - All containers created and at least one is running.
3. Succeeded
   - All containers terminated successfully and will not restart.
4. Failed
   - All containers terminated, and **at least one container failed**.
5. Unknown
   - API server can't communicate with the kubelet on that node.

Container States inside a Pod

Each container has its own state:

1. **Waiting** → preparing to start (pulling image, setup)
2. **Running** → container is operating normally
3. **Terminated** → completed or failed (includes exit code & reason)

Restart Policies

Defined in pod spec → affects what Kubernetes does if containers exit:

- **Always** → restart regardless of success/failure
- **OnFailure** → restart only if container exits with failure
- **Never** → never restart the container

(Example: batch jobs → Never/OnFailure, web servers → Always)

Pod Conditions

PodStatus contains conditions that indicate readiness:

- **PodScheduled** → pod has been assigned to a node
- **Initialized** → all init containers completed
- **ContainersReady** → all containers are ready
- **Ready** → pod is ready to serve traffic

Each condition has:

- type, status (True/False/Unknown), lastProbeTime, lastTransitionTime, reason, message

✅ 3. Identify how resources can be allocated & restricted per-container

Kubernetes allows you to control **CPU and memory** for each container using:

Resource Requests

- The **minimum guaranteed** amount of CPU/memory the container needs.
- Scheduler uses it to place pods on nodes.
- Example:

resources:

-   requests:
-     memory: "50Mi"
-

Resource Limits

- The **maximum** amount of CPU/memory a container is allowed to use.
- Prevents resource hogging.
- If a container exceeds:
  - Memory limit → OOMKill (Out Of Memory Kill)
  - CPU limit → throttled

Example:

```
resources:
  limits:
    memory: "250Mi"
```

Behaviour Example

- If container tries to allocate **more memory than the limit**, Kubernetes kills it:
  - Pod status becomes **CrashLoopBackOff**
  - Reason: **OOMKilled**

✅ 5. Kubernetes Namespaces — Summary

What is a Namespace?

A **namespace** is a logical partition inside a Kubernetes cluster that allows multiple *virtual clusters* within the same physical cluster.

Key Points

- They **scope** resource names (names must be unique inside a namespace).
- Resources cannot belong to more than one namespace.
- Namespaces cannot be nested.
- Best used when:
  - Many users
  - Spread across multiple teams/projects
    (NOT needed for small teams.)

When *not* to use namespaces

- Don’t create namespaces for different software versions → use **labels** instead.

Built-in Kubernetes Namespaces

1. **default** — for objects without a specified namespace
2. **kube-system** — for system/cluster components
3. **kube-public** — publicly readable by all (even unauthenticated users)
4. **kube-node-lease** — stores node heartbeat info for health checks

Commands

- List namespaces:
  kubectl get namespace

Naming caution

- Don’t start namespace names with **kube-** (reserved for system use)

✅ 6. Limit Ranges — Summary (Kubernetes Policies)

What is a Kubernetes Policy?

A set of rules enforced by the cluster to control how resources behave.

LimitRange (Policy Type)

A LimitRange limits:

- Minimum and maximum **memory/cpu** a container can use.
- Can also set **default requests/limits** when not defined in a pod.

Why needed?

By default, containers have **no resource constraints**, which risks cluster exhaustion.

How LimitRange works

Placed **inside a namespace** → applies to all pods in that namespace.

Kubernetes checks:

1. If pod does NOT specify memory requests/limits → apply defaults from LimitRange.
2. If pod specifies its own values → Kubernetes verifies:
    - request ≥ min
    - limit ≤ max
    - otherwise: Forbidden error

Example Behavior

- Pod without memory limits → gets default limit and request from LimitRange
- Pod with valid custom limits → allowed
- Pod exceeding max (e.g., 800Mi > 750Mi) → **creation denied**

✅ 7. ReplicaSets — Summary

What is a ReplicaSet?

A controller that ensures a stable number of identical pod replicas are running at all times.

Functions

- Self-healing (recreate pods if deleted)
- Scaling (increase/decrease replicas)
- Maintain desired number of pods

Key Concepts

- Part of the apps/v1 API group
- Used under the hood by **Deployments**
- Has:
    - **selector** → match labels to identify pods
    - **replicas** → number of desired pod copies
    - **template** → pod specification

ownerReferences

Each pod created by a ReplicaSet has an **ownerReference** linking it back to its ReplicaSet.

Acquiring pods

If a pod:

- has no ownerReference
- AND matches the ReplicaSet's selector
  → the ReplicaSet will **adopt** it.

Usage

We rarely manage ReplicaSets directly—Deployments typically handle them.

✅ 8. DaemonSets — Summary

What is a DaemonSet?

A controller that ensures **one pod per node** (or more, depending on selectors).

Why used?

To run system-level services on each node such as:

- Logging agents
- Monitoring agents
- Storage daemons

Behavior

- When a node is added → a pod is created on that node
- When a node is removed → the pod is deleted
- When DaemonSet is deleted → its pods are removed (unless cascade=false)

Accessing DaemonSet pods

Methods:

- **Push model** (pods push logs to a central service)
- NodeIP + HostPort
- DNS (headless service)
- **Regular service** (load balances randomly across nodes)

YAML requirements

- apiVersion: **apps/v1**
- kind: **DaemonSet**
- metadata: name, labels
- spec includes:
  - selector
  - template (pod spec)
- pod RestartPolicy must be **Always**

Scheduling

DaemonSet pods are **not created by the default scheduler**—they're created by the DaemonSet controller.

✅ 9. Kubernetes Deployments — Summary

What is a Deployment?

A higher-level controller used for declarative updates of pods and ReplicaSets.

Key Uses

- Rollout new ReplicaSets
- Roll back to previous versions
- Scale pods up/down
- Update pod template specs (new desired state)
- Clean up old ReplicaSets

Example use cases

- Rolling updates (zero-downtime)
- Versioned rollbacks
- Scaling to meet traffic demand

YAML Anatomy

- apiVersion: **apps/v1**
- kind: Deployment
- metadata: name
- spec:
  - replicas: number of desired pods
  - selector: matchLabels
  - template:
    - metadata: labels
    - spec: container definitions (image, ports, etc.)

Relationship

Deployment → manages ReplicaSet → manages Pods

✅ 11. Kubernetes Services — Key Concepts

Why Services?

- Pods are *ephemeral* → their IPs change whenever pods restart.
- You cannot rely on pod IPs.
- A **Service** gives a **stable virtual IP (ClusterIP)** that points to a logical set of pods.

🔷 What is a Service?

A Service is:

- An **abstraction** that groups pods using **label selectors.**
- Exposes these pods as a **network service.**
- Ensures consistent connectivity even when pods die and recreate.

Example:
Frontend → Service → Backend pods
(Frontend never needs to know pod IPs.)

🔷 How Kubernetes Service Discovery Works

1. DNS-based discovery (recommended)
   - Internal DNS (like **CoreDNS**) watches the API server.
   - Creates DNS records for each Service.
   - Every Pod can access services by name:
     `my-service.my-namespace.svc.cluster.local`
2. Environment variables (older method)
   - Kubelet injects env vars for each service into pods.
   - Works only if Service exists *before* Pods are created.

🔷 Service YAML Basics

```
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: svcApp
  ports:
    - port: 80        # service port inside cluster
      targetPort: 9000  # container port
      protocol: TCP
```

🔷 Services Without Selectors

Used when:

- Backend is **external DB** or external service.
- Service points to another namespace/cluster.
- Partial migration (some pods in Kubernetes, some outside).

Here YOU manually create **Endpoints** object.

🔷 Headless Services (clusterIP: None)

Used when:

- You don't need load balancing
- You want DNS to return **pod IPs directly**
- Useful for:
  - StatefulSets
  - Databases (e.g., Cassandra, ZooKeeper)

🔷 Service Types

Type	Description	Use Case
ClusterIP (default)	Internal-only access	Backend services
NodePort	Accessible on `<NodeIP>:<NodePort>`	Simple external access
LoadBalancer	Cloud provider LB	Public access in cloud
ExternalName	CNAME to external DNS	External service reference

✅ 12. Exploring Services in Action — Demo Summary

Goal:

Create a **NodePort Service** exposing an nginx pod.

Steps:

1. Created demo namespace:

```
kubectl create ns demo
```

2.

3. Created a Pod from YAML (`nginx:1.19.0`).
4. Exec into Pod:

```
kubectl exec nginx-pod -n demo -it -- /bin/sh
```

5.

   - Custom `index.html` was added inside container.

6. Applied the Service YAML:

   `kind: Service`

7. type: NodePort
8. ports:
9.   - port: 80
10.   targetPort: 80
11.   nodePort: 31000
12. selector:
13.   app: nginx
14.

15. Found Node IP:

   kubectl get nodes -o wide

16.

17. Accessed service in browser:

   `http://<NodeIP>:31000`

18.

Result: Custom Nginx page appears using NodePort access.

✅ 13. Identifying Ingress — Manage External Traffic

What is an Ingress?

- A Kubernetes object that routes external HTTP/HTTPS traffic into the cluster.
- Uses **Ingress rules** to route URLs → Services.
- Needs an **Ingress Controller** to function (e.g., Nginx Ingress Controller).

🔷 Ingress Requirements

1. **Ingress Controller**
   In minikube:

   ```
   minikube addons enable ingress
   ```
2.

3. **Ingress Object**
   Created using YAML manifest.

🔷 Demo Architecture

Two pods + services:

Pod	Service	Message	URL Path
app 1	app1-svc	greetings from app1	/path1
app 2	app2-svc	greetings from app2	/path2

🔷 Ingress YAML Example

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-ingress
spec:
```

```
rules:
- host: domain.local
  http:
    paths:
    - path: /path1
      pathType: Prefix
      backend:
        service:
          name: app1-svc
          port:
            number: 5678
    - path: /path2
      pathType: Prefix
      backend:
        service:
          name: app2-svc
          port:
            number: 5678
```

🔷 Checking Ingress

Watch for IP assignment:

```
kubectl get ing simple-ingress -o wide -w
```

Minikube ingress exposes:

```
minikube ip
```

Map domain locally (in Windows hosts file):

```
172.17.20.26   domain.local
```

🔷 Access Ingress

In browser:

- http://domain.local/path1 → **app1 response**
- http://domain.local/path2 → **app2 response**

🎯 Final Key Differences to Remember

Feature	Service	Ingress
Purpose	Internal service discovery & stable networking	HTTP routing from outside
Gateway?	No	Yes (reverse proxy)
Works on	L3/4 (ports)	L7 (HTTP/HTTPS)
Needs controller?	No	Yes
Supports path-based routing?	No	Yes