A pod is a collection of containers and its storage inside a node of a Kubernetes cluster. It is possible to create a pod with multiple containers inside it. For example, keeping a database container and data container in the same pod.

Types of Pod:

There are two types of Pods −

- Single container pod
- Multi container pod

Point to Remember:

- Each Pod is assigned a unique IP address. Every container in a Pod shares the network namespace, including the IP address and network ports.
- Containers *inside a Pod* can communicate with one another using **localhost**.
- A Pod can specify a set of shared storage *volumes*. All containers in the Pod can access the shared volumes, allowing those containers to share data.
- Volumes also allow persistent data in a Pod to survive in case one of the containers within needs to be restarted.
- Controllers use Pod Templates to make actual pods.

Note: The containers in a Pod are automatically co-located and co-scheduled on the same physical or virtual machine in the cluster.

POD Lifecycle:

Value	Description
Pending	The Pod has been accepted by the Kubernetes system, but one or more of the Container images has not been created. This includes time before being scheduled as well as time spent downloading images over the network, which could take a while.

Value	Description
Running	The Pod has been bound to a node, and all of the Containers have been created. At least one Container is still running, or is in the process of starting or restarting.
Succeeded	All Containers in the Pod have terminated in success, and will not be restarted.
Failed	All Containers in the Pod have terminated, and at least one Container has terminated in failure. That is, the Container either exited with non-zero status or was terminated by the system.
Unknown	For some reason the state of the Pod could not be obtained, typically due to an error in communicating with the host of the Po

Pod conditions

A Pod has a PodStatus, which has an array of PodConditions through which the Pod has or has not passed. Each element of the PodCondition array has six possible fields:

- The **lastProbeTime** field provides a timestamp for when the Pod condition was last probed.
- The **lastTransitionTime** field provides a timestamp for when the Pod last transitioned from one status to another.
- The **message** field is a human-readable message indicating details about the transition.
- The **reason** field is a unique, one-word, CamelCase reason for the condition's last transition.
- The **status** field is a string, with possible values "**True**", "**False**", and "**Unknown**".
- The **type** field is a string with the following possible values:
- **PodScheduled**: the Pod has been scheduled to a node;
- **Ready**: the Pod is able to serve requests and should be added to the load balancing pools of all matching Services;
- **Initialized**: all init containers have started successfully;
- **Unschedulable**: the scheduler cannot schedule the Pod right now, for example due to lacking of resources or other constraints;
- **ContainersReady**: all containers in the Pod are ready.

Container probes

A Probe is a diagnostic performed periodically by the kubelet on a Container. To perform a diagnostic, the kubelet calls a Handler implemented by the Container. There are three types of handlers:

- ExecAction: Executes a specified command inside the Container. The diagnostic is considered successful if the command exits with a status code of 0.
- TCPSocketAction: Performs a TCP check against the Container's IP address on a specified port. The diagnostic is considered successful if the port is open.
- HTTPGetAction: Performs an HTTP Get request against the Container's IP address on a specified port and path. The diagnostic is considered successful if the response has a status code greater than or equal to 200 and less than 400.

Each probe has one of three results:

- Success: The Container passed the diagnostic.
- Failure: The Container failed the diagnostic.
- Unknown: The diagnostic failed, so no action should be taken.

The kubelet can optionally perform and react to two kinds of probes on running Containers:

- **livenessProbe**: Indicates whether the Container is running. If the liveness probe fails, the kubelet kills the Container, and the Container is subjected to its restart policy. If a Container does not provide a liveness probe, the default state is **Success**.
- **readinessProbe**: Indicates whether the Container is ready to service requests. If the readiness probe fails, the endpoints controller removes the Pod's IP address from the endpoints of all Services that match the Pod. The default state of readiness before the initial delay is **Failure**. If a Container does not provide a readiness probe, the default state is **Success**

When should you use liveness or readiness probes?

If the process in your Container is able to crash on its own whenever it encounters an issue or becomes unhealthy, you do not necessarily need a liveness probe; the kubelet will automatically perform the correct action in accordance with the Pod's **restartPolicy**.

If you'd like your Container to be killed and restarted if a probe fails, then specify a liveness probe, and specify a **restartPolicy** of Always or OnFailure.

Note:

If you'd like to start sending traffic to a Pod only when a probe succeeds, specify a readiness probe. In this case, the readiness probe might be the same as the liveness probe, but the existence of the readiness probe in the spec means that the Pod will start without receiving any traffic and only start receiving traffic after the probe starts succeeding.

Restart policy

- A PodSpec has a **restartPolicy** field with possible values Always, OnFailure, and Never. The default value is Always. **restartPolicy** applies to all Containers in the Pod.
- **restartPolicy** only refers to restarts of the Containers by the kubelet on the same node.
- Exited Containers that are restarted by the kubelet are restarted with an exponential back-off delay (10s, 20s, 40s …) capped at five minutes, and is reset after ten minutes of successful execution.

State Examples:

Pod is running and has one Container. Container exits with success.

- Log completion event.
- If **restartPolicy** is:
- Always: Restart Container; Pod **phase** stays Running.
- OnFailure: Pod **phase** becomes Succeeded.
- Never: Pod **phase** becomes Succeeded.

Pod is running and has one Container. Container exits with failure.

- Log failure event.
- If **restartPolicy** is:
- Always: Restart Container; Pod **phase** stays Running.

- OnFailure: Restart Container; Pod **phase** stays Running.
- Never: Pod **phase** becomes Failed.

Volumes:

- On-disk files in a Container are ephemeral, which presents some problems for non-trivial applications when running in Containers.
- First, when a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state. Second, when running Containers together in a **Pod** it is often necessary to share files between those Containers.

The Kubernetes **Volume** abstraction solves both of these problems.
Types of Volumes

Kubernetes supports several types of Volumes:

- awsElasticBlockStore
- azureDisk
- azureFile
- cephfs
- configMap
- csi
- downwardAPI
- emptyDir
- fc (fibre channel)
- flocker
- gcePersistentDisk
- gitRepo (deprecated)
- glusterfs
- hostPath
- iscsi
- local
- nfs
- persistentVolumeClaim

- projected
- portworxVolume
- quobyte
- rbd
- scaleIO
- secret
- storageos
- vsphereVolume

Sample POD using YML:

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

Kubernetes RestApI :

https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.12/#networkpolicy-v1-networking-k8s-io

Steps:

Create Deployment :

1. kubectl run demo-backend --image=demo-backend:latest \
   --port=8080 --image-pull-policy Never

Verify deployments:

2. kubectl get deployments

kubectl logs <pod id>

3. Create a service for deployment:
   kubectl expose deployment demo-backend --type=NodePort

4. kubectl get services

Call a service from minikube:

5. minikube service demo-backend

Deleting:

kubectl delete service demo-backend
kubectl delete deployment demo-backend

Using Config:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-backend
spec:
  selector:
      matchLabels:
        app: demo-backend
  replicas: 3
  template:
    metadata:
      labels:
        app: demo-backend
    spec:
      containers:
        - name: demo-backend
          image: demo-backend:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 8080
```

- `kubectl create -f backend-deployment.yaml`

Creating Service and Deployment:

```
kind: Service
apiVersion: v1
metadata:
  name: demo-frontend
spec:
  selector:
    app: demo-frontend
  ports:
  - protocol: TCP
    port: 8081
    nodePort: 30001
  type: NodePort
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: demo-frontend
spec:
  selector:
      matchLabels:
        app: demo-frontend
  replicas: 3
  template:
    metadata:
      labels:
        app: demo-frontend
    spec:
      containers:
        - name: demo-frontend
          image: demo-frontend:latest
          imagePullPolicy: Never
          ports:
            - containerPort: 8081
```