

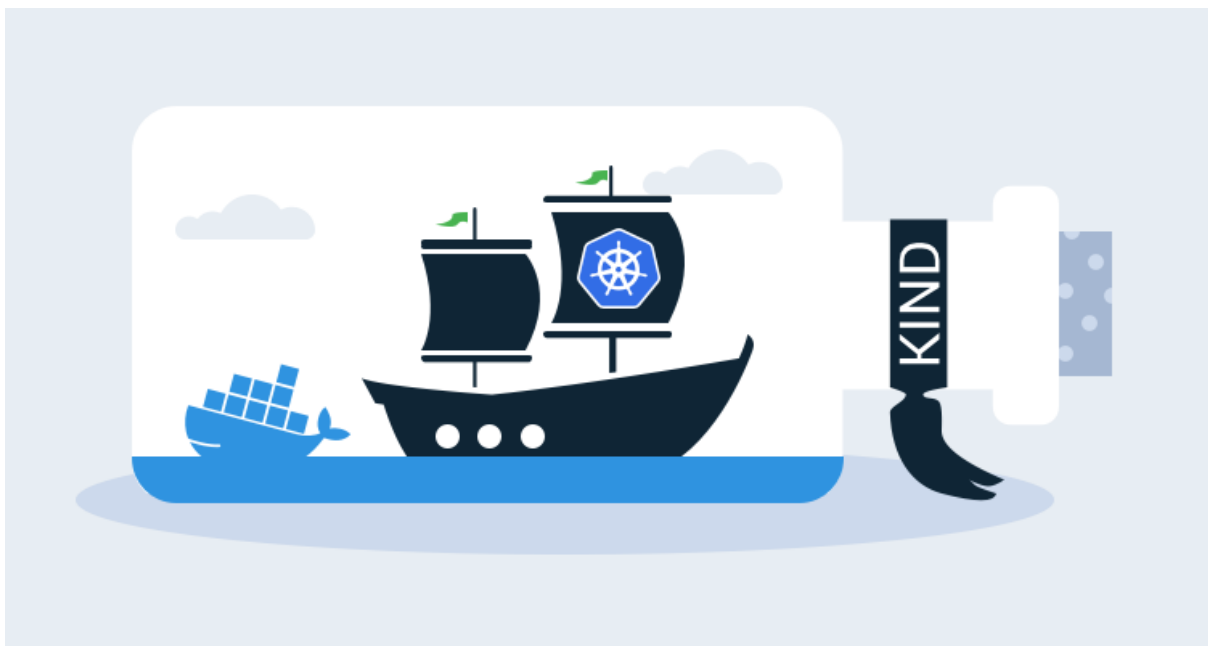
Kubernetes with KIND: Local Cluster Setup, Pods, Deployments & Advanced Features

➤ KIND

KIND means (Kubernetes IN Docker) KIND is a tool that allows you to run Kubernetes clusters locally using Docker containers as nodes. It's lightweight and primarily used for testing and local development of Kubernetes clusters.

Why Use KIND?

- Lightweight:
- Fast Setup
- Multi-Node Clusters
- Local Development



Prerequisites

Docker Installed: KIND uses Docker to create Kubernetes nodes.

- Install Docker

KIND Installed:

- Install KIND via Go or download the binary.

For windows installation command for the kind :

➤ **Choco install kind**

Creating a Kubernetes Cluster Using KIND with a Specified Image

1. Prerequisites

Before creating a cluster, ensure the following:

- Docker is installed and running.
- KIND is installed and accessible from the command line.
- Install the kubectl
- A Kubernetes image tag is known (e.g., kindest/node:v1.26.0,).

➤ The following are pre-built Kubernetes node images for KIND, matching specific Kubernetes versions. They include SHA256 digests for verification:

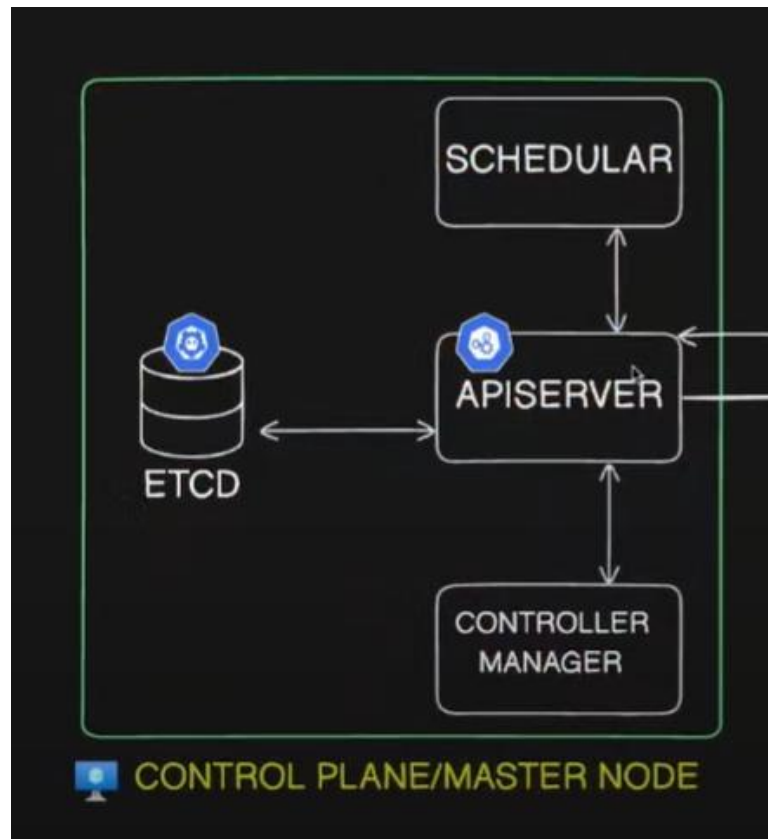
- **v1.32.0: kindest/node:v1.32.0@sha256:c48c62eac5da28cdadcf560d1d8616cfa6783b58f0d94cf63ad1bf49600cb027**
- **v1.31.4: kindest/node:v1.31.4@sha256:2cb39f7295fe7eafee0842b1052a599a4fb0f8bcf3f83d96c7f4864c357c6c30**
- **v1.30.8: kindest/node:v1.30.8@sha256:17cd608b3971338d9180b00776cb766c50d0a0b6b904ab4ff52fd3fc5c6369bf**
- **v1.29.12: kindest/node:v1.29.12@sha256:62c0672ba99a4afd7396512848d6fc382906b8f33349ae68fb1dbfe549f70dec**
- **Ensure the URL is directly referenced as the source of the images:**
- You can find the pre-built images of Kubernetes for KIND at the following URL:
<https://github.com/kubernetes-sigs/kind/releases>

2. Default Kubernetes Image

By default, KIND uses the official Kubernetes node image, which can be specified using the `--image` flag.

3. Creating a Single-Node Cluster with a Specific Image

Use the following command to create a single-node cluster using a specified Kubernetes version:



Command:

```
kind create cluster --name my-single-cluster --image  
kindest/node:v1.26.0
```

- Explanation:

- --name: Specifies the cluster name (default is kind).
- --image: Specifies the Kubernetes node image (e.g., kindest/node:v1.26.0).

```

C:\Users\penuj>kind create cluster --name my-single-cluster
Creating cluster "my-single-cluster" ...
  • Ensuring node image (kindest/node:v1.32.0) ...
  ✓ Ensuring node image (kindest/node:v1.32.0) ...
  • Preparing nodes ...
  ✓ Preparing nodes ...
  • Writing configuration ...
  ✓ Writing configuration ...
  • Starting control-plane ...
  ✓ Starting control-plane ...
  • Installing CNI ...
  ✓ Installing CNI ...
  • Installing StorageClass ...
  ✓ Installing StorageClass ...
Set kubectl context to "kind-my-single-cluster"
You can now use your cluster with:

kubectl cluster-info --context kind-my-single-cluster

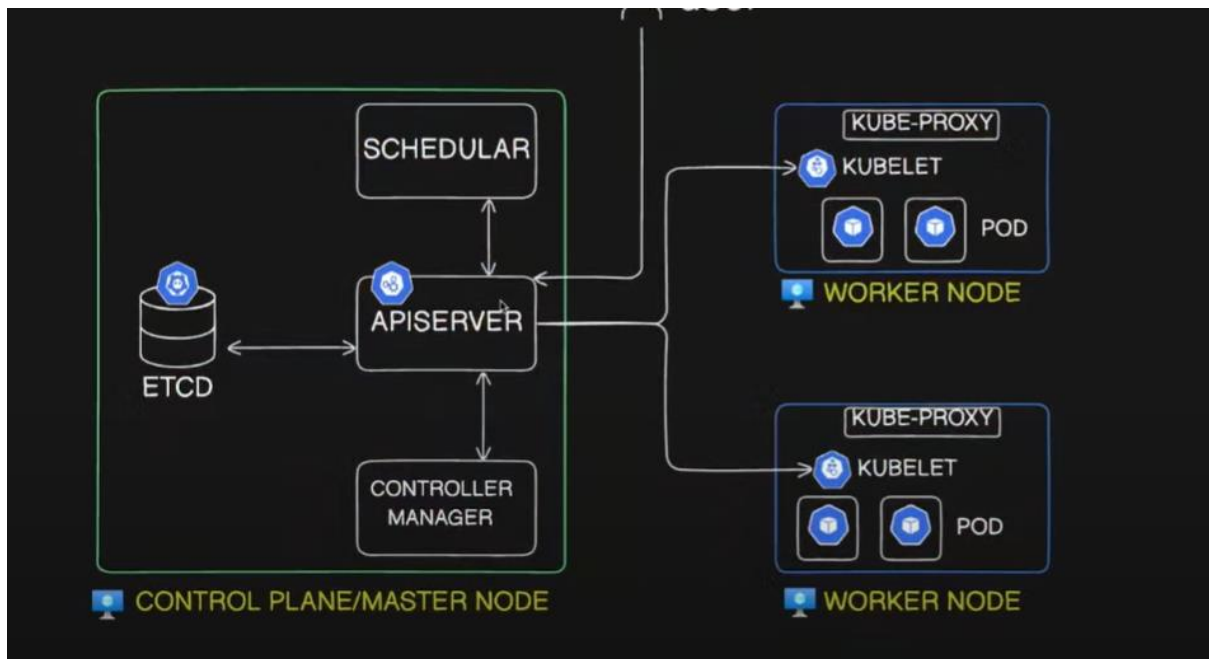
```

Check the nodes command :

➤ Kubectl get nodes

Then u will be find the only the control plane , which serves as both the control plane (master node) and the worker node

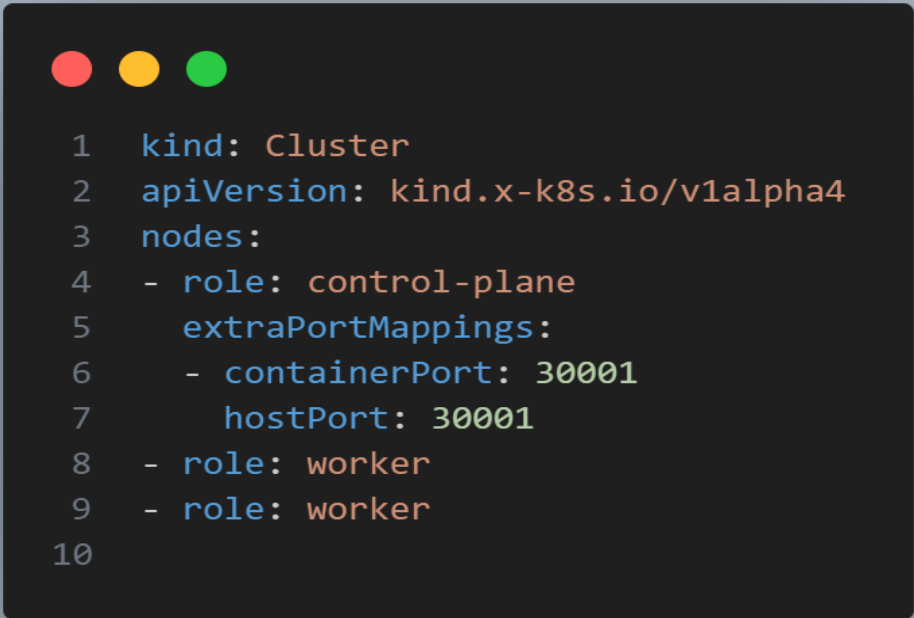
Creating a Multi-Node Cluster with a Specific Image



To create a multi-node cluster, define a custom configuration file where you specify the image for each node.

Step 1: Create a Configuration File

Create a file named `kind-multi-node-config.yaml` with the following content:



```
1 kind: Cluster
2 apiVersion: kind.x-k8s.io/v1alpha4
3 nodes:
4 - role: control-plane
5   extraPortMappings:
6   - containerPort: 30001
7     hostPort: 30001
8 - role: worker
9 - role: worker
10
```

Explanation:

- `kind: Cluster`
Specifies the resource type. In this case, it defines a KIND cluster configuration.
- `apiVersion: kind.x-k8s.io/v1alpha4`
Specifies the API version for the KIND configuration. This ensures compatibility with the KIND tool.

- nodes:
Defines the nodes in the cluster. Each node's configuration is listed under this field.

Node-Specific Fields:

1. role:

- Specifies the role of the node in the cluster.
 - control-plane: This node runs the Kubernetes API server, scheduler, and controller manager, managing the cluster.
 - worker: These nodes are used to run application workloads (pods).

2. extraPortMappings: *(only under the control-plane node)*

- Configures port forwarding between the host machine and the control-plane node container.
 - containerPort: The port inside the Kubernetes cluster (in the control-plane node).
 - hostPort: The port on the host machine mapped to the container port.

Example in this config:

- containerPort: 30001 → A service running in the cluster on port 30001 will be accessible on the host machine via localhost:30001.

Command to Create the multi node Cluster:

```
kind create cluster --name multi-cluster --image  
kindest/node:v1.26.0 --config kind-multi-node.yaml
```

Explanation of the Command:

- `--name multi-cluster`: Specifies the name of the KIND cluster (can be any valid name).
- `--config kind-multi-node.yaml`: Points to the custom configuration file defining the cluster setup.
- `image`: Specifies the Kubernetes version for the nodes.

Check the nodes command in the multi node cluster :

➤ `kubectl get nodes`

Then u will be find the control plane and the worker nodes

List Available Clusters:

➤ `kubectl config get-contexts`

Switch to a Specific Cluster:

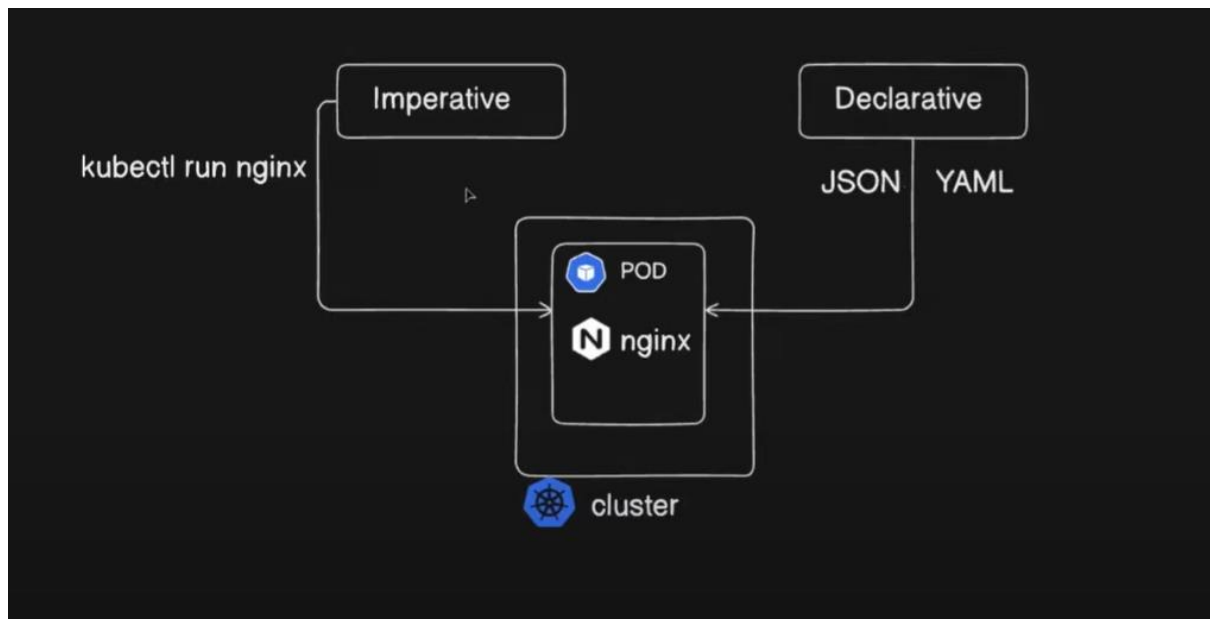
➤ `kubectl config use-context kind <cluster-name>`

Finding the Current Cluster Context:

➤ `kubectl config current-context`

=====

CREATION OF POD IN KUBERNETES IN IMPERATIVE AND DECLARATIVE WAY



Imperative Approach

In the **imperative way**, you directly execute commands to apply changes to the cluster

Characteristics:

1. Direct Action

2. No Record:

- there's no persistent configuration file

3. Quick Execution

> command for the creating a pod :

- `Kubectl run nginx-pod --image=nginx:latest`

Check the pod a

- `kubectI get pods`

Declarative Approach

In the **declarative way**, you define the desired state of your resources in configuration files (e.g., YAML or JSON) and apply them to the cluster

Characteristics:

1. **Desired State**

2. **Version Control:**

- Configurations can be stored in repositories like Git.

3. **Reproducibility>**

- Easy to reapply configurations and ensure consistency.

Example:

- Defining a pod in a YAML file (pod.yaml):



```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5  spec:
6    containers:
7      - name: nginx
8        image: nginx
9
```

Apply the file using:

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

if you want to delete the pod

* command : `kubectl delete pod <name_of_pod>`

Deployment in Kubernetes

- A Deployment in Kubernetes is an API object that provides declarative updates to Pods and ReplicaSets. It defines the desired state for your application, such as the number of replicas, container image, and configuration, and ensures that the current state matches this desired state.

Key Features

- **Self-healing:**
 - Kubernetes monitors Pods and ensures the desired number of replicas are running.
 - If a Pod crashes or becomes unresponsive, Kubernetes automatically replaces it.
- **Scaling Applications:**
 - **Horizontal Scaling:** Adjust the number of replicas by updating the replicas field in the deployment specification.
 - Scaling can be manual or automated using Horizontal Pod Autoscalers (HPA).
- **Rolling Updates:**
 - Deployments support zero-downtime updates through rolling updates.
 - The application is updated incrementally by creating new Pods with the updated specification while gradually terminating the old Pods.
 - This ensures service availability during updates.
- **Rollback Capability:**

- Deployments store the history of previous states.
- If an update fails or causes issues, you can roll back to a previous version.

Deployment YAML Example

Here's an example of a simple deployment manifest:



```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: nginx-deployment
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        app: nginx
10   template:
11     metadata:
12       labels:
13         app: nginx
14     spec:
15       containers:
16       - name: nginx
17         image: nginx:latest
18         ports:
19         - containerPort: 80
```

- **replicas:** Defines the number of Pod replicas.
- **selector:** Identifies Pods that belong to the deployment.

- **template:** Specifies the Pod configuration.

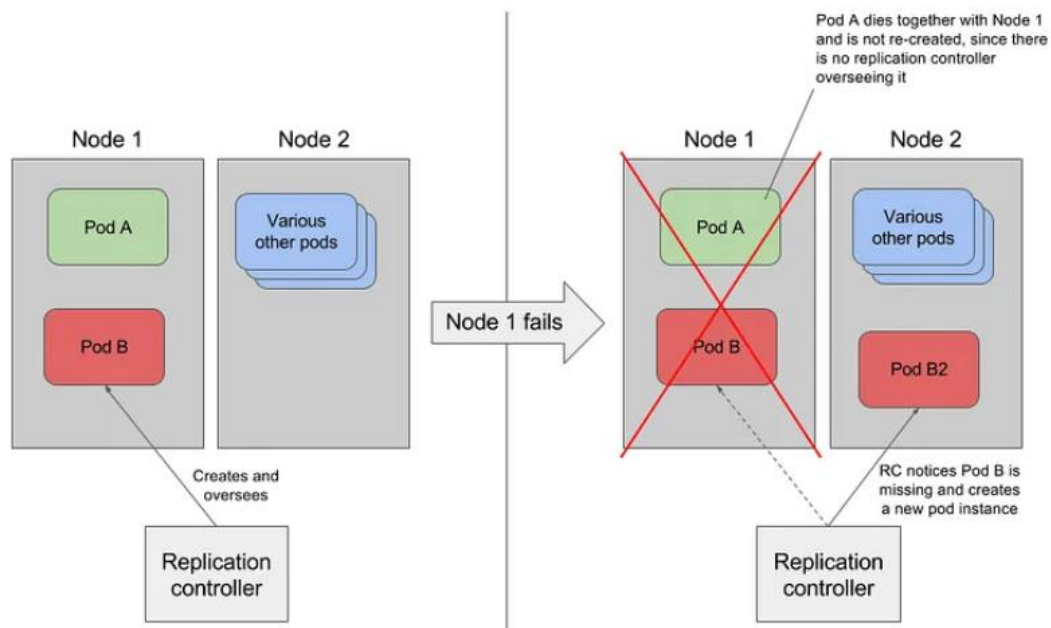
Purpose of the deployment.yaml File

The deployment.yaml file is used to define:

- The desired number of replicas (pods) for your application.
- The **container specifications** (such as images, ports).
- **Rolling update strategies** to update applications with zero downtime.
- **Pod health checks** to ensure pods are running correctly.
- **Scaling** rules to increase or decrease the number of pods based on load.

ReplicationController

A **ReplicationController** in Kubernetes ensures that a specified number of pod replicas are running at any given time. It continuously monitors the cluster and takes corrective actions to maintain the desired state, such as creating new pods or terminating excess ones.



Key Features:

- **Replication:** Maintains the desired number of pod replicas.
- **Self-healing:** Automatically replaces failed pods.
- **Declarative configuration:** Desired state is defined in a YAML or JSON manifest file.
- **Scalability:** Supports scaling up or down by changing the desired replica count.

Example YAML Manifest for a ReplicationController



```
1  apiVersion: v1
2  kind: ReplicationController
3  metadata:
4    name: my-app-rc
5    labels:
6      app: my-app
7  spec:
8    replicas: 3
9    selector:
10     matchLabels:
11       app: my-app
12   template:
13     metadata:
14       labels:
15         app: my-app
16     spec:
17       containers:
18       - name: my-app-container
19         image: nginx
20         ports:
21         - containerPort: 80
22
```

- **replicas:** Ensures 3 pods are running.
- **selector:** Identifies pods with the label app: my-app.

- **template:** Specifies the pod's configuration.

Uses of ReplicationController

1. Ensuring High Availability:

- Guarantees that a defined number of pod replicas are running.
- If a pod fails, the ReplicationController replaces it, ensuring application availability.

2. Load Distribution:

- Ensures multiple replicas of a pod are spread across the cluster.
- Improves resource utilization and ensures that workloads are balanced.

3. Fault Tolerance:

- If a node crashes or a pod is deleted, the ReplicationController recreates the missing pods to meet the desired state.

4. Scaling Applications:

- Easily scales the application up or down by adjusting the replicas field.
- This can be done manually or through automation using Horizontal Pod Autoscalers.

ReplicaSet in Kubernetes(it is the update version of the replicacontroller)

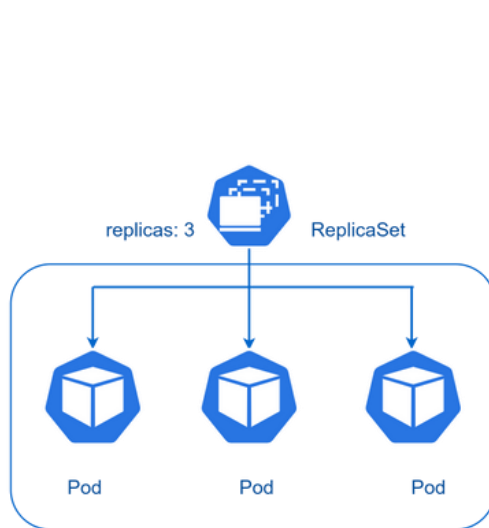


Fig: ReplicaSet

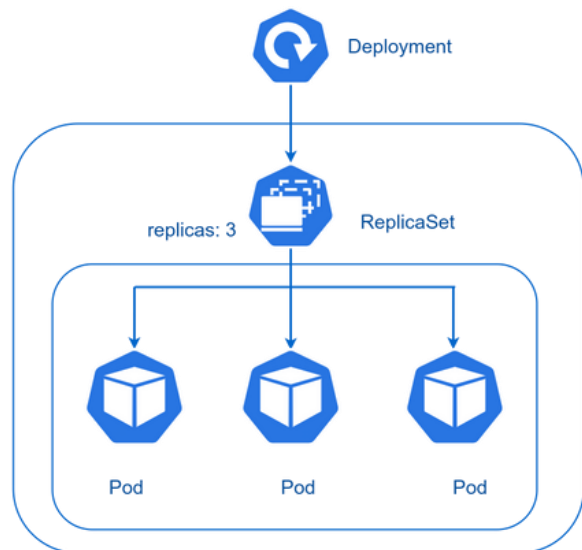


Fig: Deployment

#goglides

A **ReplicaSet** is a Kubernetes resource used to ensure a specific number of identical pod replicas are running at any given time. It provides fault tolerance, scalability, and self-healing capabilities. ReplicaSets are primarily used to maintain pod availability and manage workloads in a cluster.

Features of ReplicaSet

1. Maintains Desired Replica Count:

- Ensures the specified number of pod replicas are running at all times.
- If a pod fails or is deleted, the ReplicaSet creates a new pod to replace it.

2. Scaling:

- Can scale the number of replicas up or down by adjusting the replicas field.

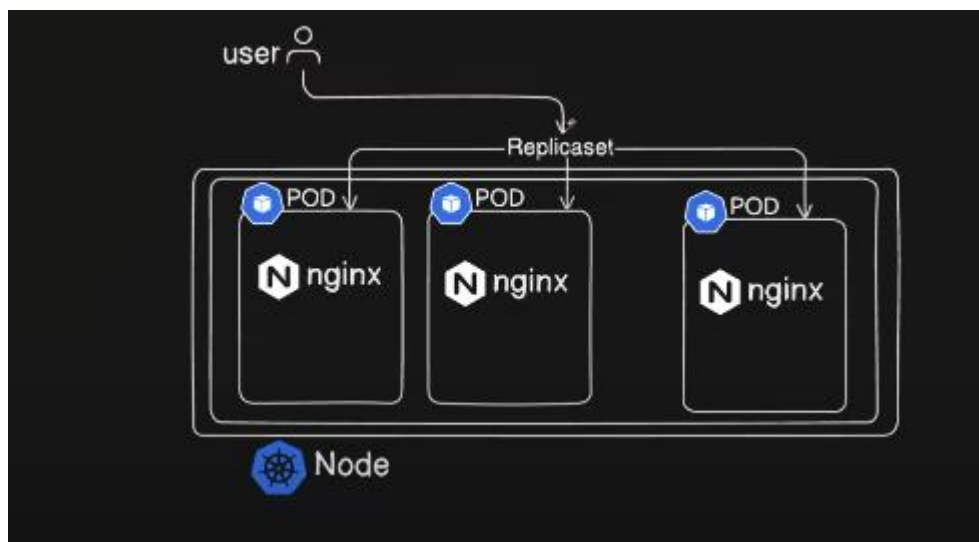
- Supports manual and automated scaling (e.g., Horizontal Pod Autoscaler).

3. Fault Tolerance:

- Automatically creates new pods to replace failed ones.
- Helps in maintaining application availability during node failures.

4. Uses Label Selectors:

- Matches pods using labels, ensuring only the intended pods are managed.
- Supports both equality-based and set-based selectors for greater flexibility.



ReplicaSet YAML Example

```
1  apiVersion: apps/v1
2  kind: ReplicaSet
3  metadata:
4    name: nginx-replicaset
5    labels:
6      app: nginx
7  spec:
8    replicas: 3 # Number of pod replicas
9    selector:
10     matchLabels:
11       app: nginx # Selector to identify pods managed by this ReplicaSet
12  template:
13     metadata:
14       labels:
15         app: nginx # Labels for the pods
16     spec:
17       containers:
18         - name: nginx
19           image: nginx:latest # Specify the Nginx image version
20           ports:
21             - containerPort: 80 # Expose port 80
22
```

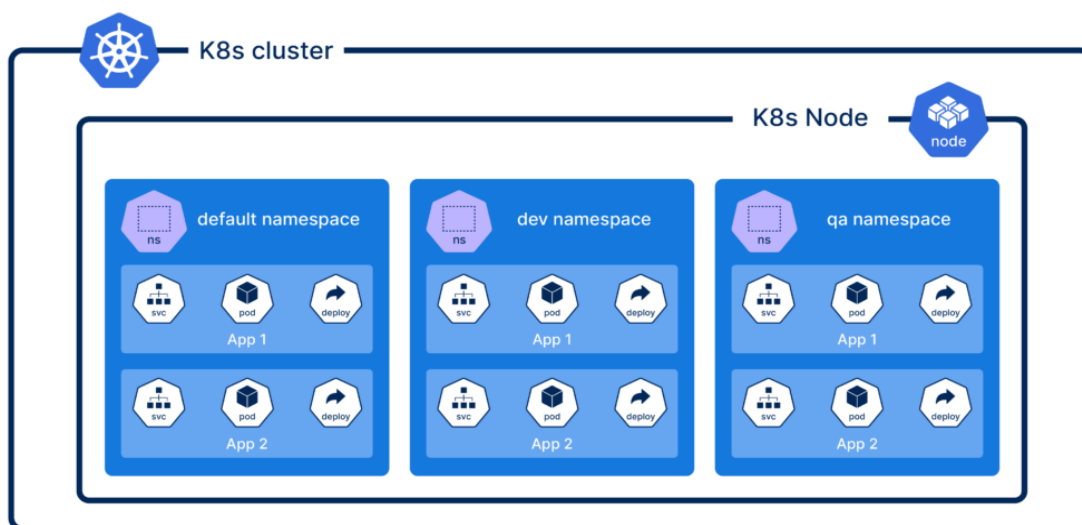
Explanation:

- **replicas**: Specifies the desired number of pod replicas to maintain.
- **selector**: Matches the pods with specified labels to be managed by the ReplicaSet.
- **template**: Defines the pod's metadata and container specifications.

Namespaces in Kubernetes

Namespaces in Kubernetes provide a mechanism for isolating resources within a cluster. They allow you to organize and manage workloads by dividing the cluster into virtual sub-clusters. This is especially useful in multi-tenant environments, where different teams or projects share the same cluster but need separate environments.

Kubernetes - Namespaces



Default Namespaces in Kubernetes


1. **default:**
The default namespace for resources if no namespace is specified.
2. **kube-system:**
Reserved for Kubernetes system components like the API server and scheduler.
3. **kube-public:**
A namespace visible to all users, including

unauthenticated users. Used for publicly accessible resources.

4. kube-node-lease:

Used for node heartbeats. It improves the performance of the node controller in large clusters.

Example Deploy a Pod in the Namespace



```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5    namespace: test-environment
6  spec:
7    containers:
8      - name: nginx
9        image: nginx:latest
10
```

Commands for Working with Namespaces

1. List All Namespaces:

```
kubectl get namespaces
```

2. Create a Namespace:

```
kubectl create namespace <namespace-name>
```

3. Delete a Namespace:

```
kubectl delete namespace <namespace-name>
```

4. View Namespace Details:

```
kubectl describe namespace <namespace-name>
```

5. Set a Default Namespace for a Context:

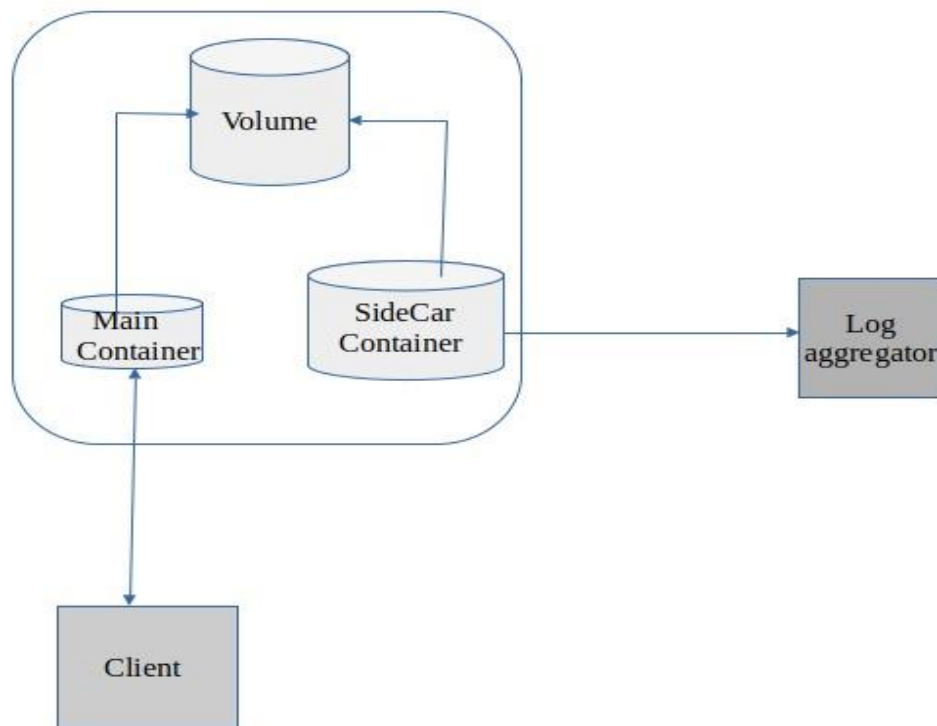
```
kubectl config set-context --current --  
namespace=<namespace-name>
```

6. Check Resources in a Namespace:

```
kubectl get all -n <namespace-name>
```

Multi-Container Pods in Kubernetes

In Kubernetes, a **pod** is the smallest deployable unit and can contain one or more containers. Multi-container pods allow containers within the same pod to share resources such as network and storage and work together to accomplish a task. Two common patterns for multi-container pods are **sidecar containers** and **init containers**.



- The *Log Aggregator* then processes and stores these logs for analysis or monitoring purposes.
- A **volume** is a directory or file system that exists outside of a container's lifecycle. It can be mounted into one or more containers, allowing them to read or write data

1. Sidecar Containers

Sidecar containers are the secondary containers that run along with the main application container within the same [Pod](#). These containers are used to enhance or to extend the functionality of the primary *app container* by providing additional services, or functionality such as logging, monitoring, security, or data synchronization, without directly altering the primary application code.

Typically, you only have one app container in a Pod. For example, if you have a web application that requires a local webserver, the local webserver is a sidecar and the web application itself is the app container.

Example application

Example of a Deployment with two containers, one of which is a sidecar:

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: myapp
5    labels:
6      app: myapp
7  spec:
8    replicas: 1
9    selector:
10     matchLabels:
11       app: myapp
12   template:
13     metadata:
14       labels:
15         app: myapp
16     spec:
17       containers:
18         - name: myapp
19           image: alpine:latest
20           command: ['sh', '-c', 'while true; do echo "logging" >> /opt/logs.txt; sleep 1; done']
21           volumeMounts:
22             - name: data
23               mountPath: /opt
24       initContainers:
25         - name: logshipper
26           image: alpine:latest
27           restartPolicy: Always
28           command: ['sh', '-c', 'tail -F /opt/logs.txt']
29           volumeMounts:
30             - name: data
31               mountPath: /opt
32       volumes:
33         - name: data
34           emptyDir: {}
35
```

Explanation

➤ Init Container:

- **Purpose:** The logshipper init container simulates initialization by monitoring (tail -F) the logs.txt file located in the shared volume.
- **Execution:** Runs before the main container and completes its task to ensure that the logging mechanism is set up.
- **Shared Volume:** Uses a shared emptyDir volume (/opt) to exchange data with the main container.

➤ Main Container:

- **Purpose:** The myapp container generates logs by appending the string logging to the logs.txt file every second.
- **Execution:** Starts only after the init container completes successfully.
- **Shared Volume:** Writes logs to the same shared emptyDir volume (/opt).

➤ Sequential Execution:

- The init container (logshipper) must finish before the main container (myapp) begins execution, ensuring the environment is prepared correctly

Commands to Deploy and Test

Create the Deployment

1. Save the YAML file as multi-container-deployment.yaml.

2. Apply the file to your Kubernetes cluster:

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

➤ Check Deployment Status

- Verify that the deployment is created and running:

- `kubectl get deployments`

```
kubectl get pods
```

➤ Describe the Pod

- Inspect the pod details:

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

➤ Access Logs

- View logs from the myapp container:

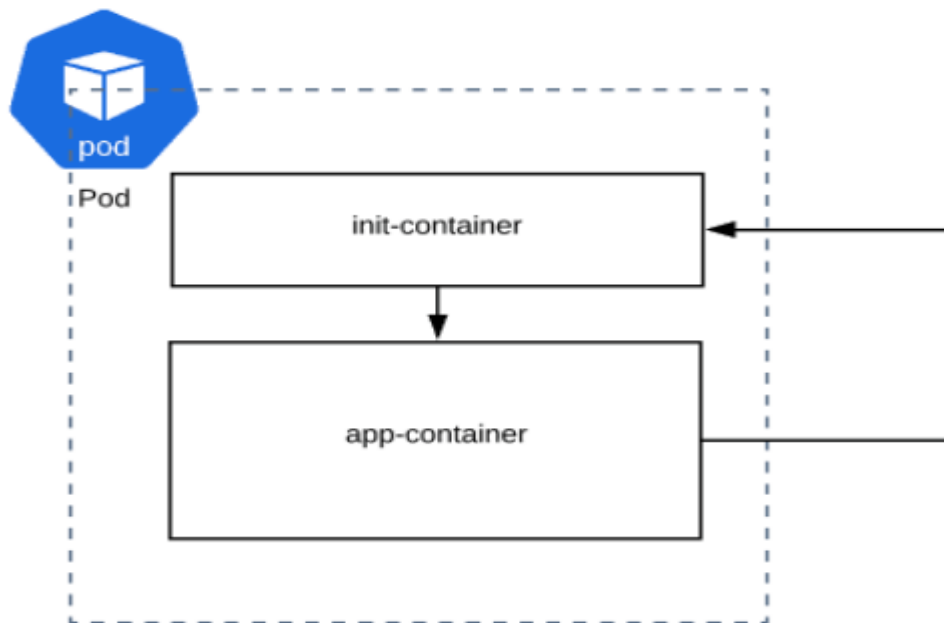
➤ `kubectl logs <pod-name> -c myapp`

- View logs from the logshipper init container:

```
kubectl logs <pod-name> -c logshipper
```

Init Containers in Kubernetes

Init containers in Kubernetes are specialized containers that run before application containers within a pod. They are used for setting up the environment, ensuring dependencies, or performing one-time initialization tasks that the main containers depend on.



Key Features of Init Containers

1. Run to Completion:

- Unlike application containers, init containers run to completion before the main containers start.

2. Sequential Execution:

- Init containers are executed one after the other in the order they are defined in the pod specification.

3. Dependency Setup:

- They can perform tasks such as waiting for a service to become available, running database migrations, or pulling configuration files.

4. Failure Handling:

- If an init container fails, the pod is not started. Kubernetes will keep restarting the failing init container until it succeeds unless the pod's restartPolicy is set to Never.

Example Pod with Init Container

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: pod-with-init
5  spec:
6    initContainers:
7      - name: init-container
8        image: busybox
9        command: ["sh", "-c", "echo 'Initializing...' && sleep 10 && echo 'Initialization complete'"]
10       volumeMounts:
11         - name: shared-data
12           mountPath: /mnt/shared
13     containers:
14       - name: main-container
15         image: nginx:latest
16         ports:
17           - containerPort: 80
18         volumeMounts:
19           - name: shared-data
20             mountPath: /usr/share/nginx/html
21     volumes:
22       - name: shared-data
23         emptyDir: {}
24
```

Explanation

1. Init Container:

- Runs a script to simulate initialization (e.g., setting up an environment or preparing data).
- Writes output to a shared volume (/mnt/shared).

2. Main Container:

- Serves static files using Nginx.
- Reads data from the shared volume (/usr/share/nginx/html) prepared by the init container.

3. Sequential Execution:

- The init container must complete successfully before the main container starts.

➤ Benefits of Init Containers

1. Environment Setup:

- Perform tasks like configuring files or setting up directories before the application starts.

2. Lightweight Design:

- Enable modular and reusable container designs by separating initialization logic from the main application.

3. Improved Reliability:

- Ensure that all dependencies are ready before starting the application containers.

Monitoring Init Containers

Commands

1. Deploy the Pod:

```
kubectl apply -f <filename>
```

2. Check Pod Status:

```
kubectl get pods
```

3. Describe the Pod:

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

➤ Differences from sidecar containers

Init containers run and complete their tasks before the main application container starts. Unlike [sidecar containers](#), init containers are not continuously running alongside the main containers.

Init containers run to completion sequentially, and the main container does not start until all the init containers have successfully completed.

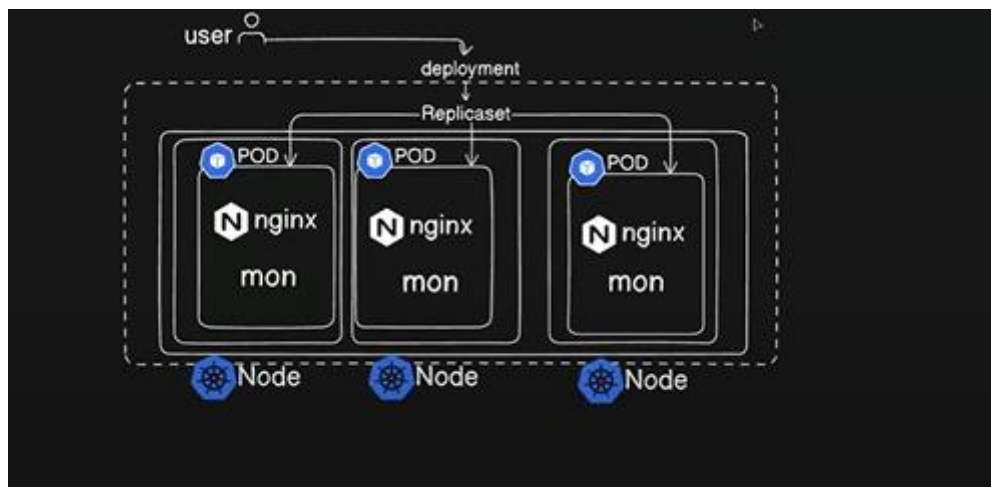
Init containers share the same resources (CPU, memory, network) with the main application containers but do not interact directly with them. They can, however, use shared volumes for data exchange.

Kubernetes DaemonSets, Jobs, and CronJobs

1. DaemonSets

A **DaemonSet** ensures that a specific pod runs on **all or a subset of nodes** in a Kubernetes cluster. It is useful for

deploying node-level services, such as logging agents, monitoring agents, or network services.




Key Characteristics:

- Ensures one copy of a pod per node.
- Automatically creates pods on new nodes when they are added to the cluster.
- Deletes pods from nodes when they are removed from the cluster.

Use Cases:

- Monitoring tools
- Networking agents
- Example DaemonSet YAML:



```
1  apiVersion: apps/v1
2  kind: DaemonSet
3  metadata:
4    name: fluentd-daemonset
5    namespace: kube-system
6  spec:
7    selector:
8      matchLabels:
9        name: fluentd
10   template:
11     metadata:
12       labels:
13         name: fluentd
14     spec:
15       containers:
16       - name: fluentd
17         image: fluent/fluentd:v1.14
18         resources:
19           limits:
20             memory: 200Mi
21             cpu: 100m
```

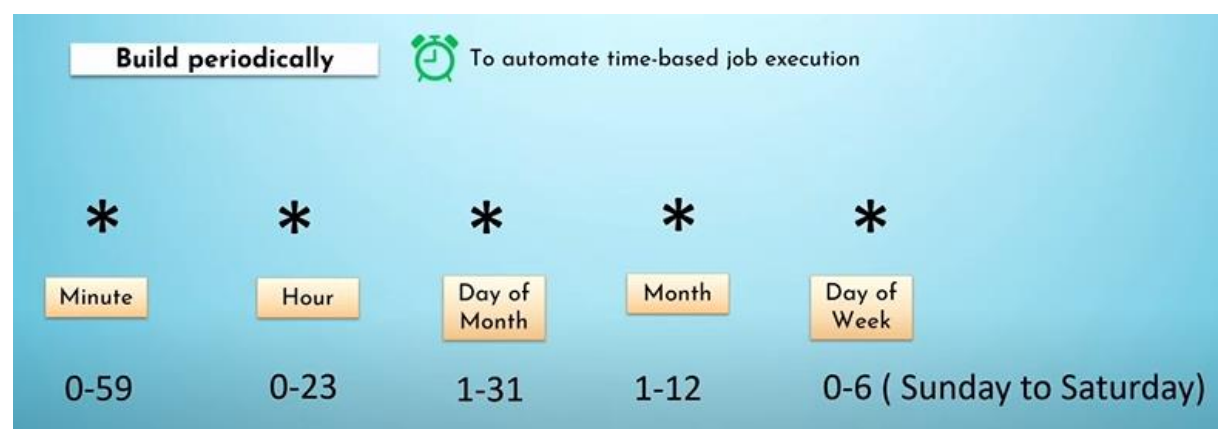
Commands:

- Create a DaemonSet:
 - `kubectl apply -f daemonset.yaml`
- Get the list of DaemonSets:
 - `kubectl get daemonsets`

- Describe a DaemonSet:
 - `kubectl describe daemonset fluentd-daemonset`
- Delete a DaemonSet:
 - `kubectl delete -f daemonset.yaml`

CronJobs

A **CronJob** in Kubernetes is used for **scheduled tasks**. It is like a Job but runs periodically based on a specified schedule using **Cron syntax**.



Key Characteristics:

- Automates recurring tasks.
- Uses Cron expressions for scheduling.
- Creates Jobs at the specified time intervals.
- Useful for tasks like backups, report generation, or sending periodic emails.

Cron Expression Syntax:

- Format: * * * * *
- Minute (0-59)
- Hour (0-23)
- Day of the month (1-31)
- Month (1-12)
- Day of the week (0-7, where 0 and 7 represent Sunday)

Use Cases:

- Running a periodic database backup.
- Cleaning up temporary files.
- Sending daily reports.

Example CronJob YAML:



```
1  apiVersion: batch/v1
2  kind: CronJob
3  metadata:
4    name: example-cronjob
5  spec:
6    schedule: "*/5 * * * *" # Every 5 minutes
7    jobTemplate:
8      spec:
9        template:
10         spec:
11         containers:
12         - name: example-task
13           image: busybox
14           command: ["/bin/sh", "-c", "echo Hello CronJob! && date"]
15         restartPolicy: OnFailure
```

Commands:

- Create a CronJob:

- `kubectl apply -f cronjob.yaml`

- Get the list of CronJobs:

- `kubectl get cronjobs`

- View the jobs triggered by a CronJob:

- `kubectl get jobs`

- Describe a CronJob:

- `kubectl describe cronjob example-cronjob`

- Delete a CronJob:

- `Kubectl delete -f cronjob.yaml`


Static Pods

Static Pods are Kubernetes pods that are **not managed by the Kubernetes API server** directly. Instead, they are created and managed by the **kubelet** on each node. These pods are useful for critical system-level workloads or for running processes that must always exist on a specific node.

How Static Pods Work

- Static Pods are defined in a configuration file directly on the node (not in the Kubernetes control plane).
- The **kubelet** reads these pod definitions from a specified directory on the node (e.g., /etc/kubernetes/manifests).
- The kubelet is responsible for creating, monitoring, and restarting these pods if they fail.
- **Static Pods do not go through the scheduler**, so they will always run on the node where they are defined.

Create a Pod YAML File



```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: static-pod-example
5    labels:
6      app: static-example
7  spec:
8    containers:
9      - name: nginx
10        image: nginx:1.21
11        ports:
12          - containerPort: 80
13
```

Commands:

- View static pods:

kubectl get pods -o wide

- Delete static pod: Delete the YAML file from the --pod-manifest-path directory.

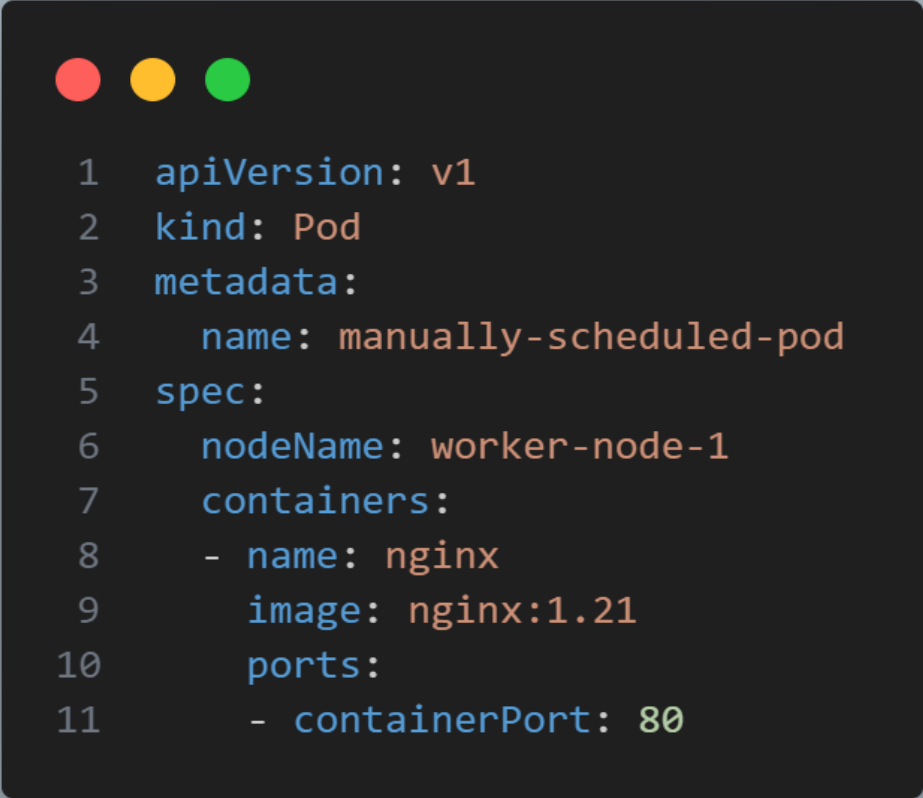
➤ Manual Scheduling

Manual scheduling allows you to assign a pod to a specific node without using a scheduler.

Key Steps:

1. Use the nodeName field in the pod specification to specify the node

Example Manual Scheduling YAML:



```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: manually-scheduled-pod
5  spec:
6    nodeName: worker-node-1
7    containers:
8      - name: nginx
9        image: nginx:1.21
10       ports:
11         - containerPort: 80
```

Commands:

- Create the pod:

- `kubectl apply -f pod.yaml`

- **Verify pod placement:**

- `kubectl get pods -o wide`

6. Labels and Selectors

Labels:

Labels are key-value pairs attached to Kubernetes objects like pods, nodes, or services. They provide identifying metadata.

Selectors:

Selectors are used to query and filter Kubernetes objects based on their labels.

Example Labels:



```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: labeled-pod
5    labels:
6      app: frontend
7      environment: production
8  spec:
9    containers:
10     - name: nginx
11       image: nginx:1.21
```

Example Selector in a Service:



```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: frontend-service
5  spec:
6    selector:
7      app: frontend
8      environment: production
9    ports:
10     - protocol: TCP
11       port: 80
12       targetPort: 80
```

Commands:

- Apply a label to an existing pod:
 - `kubectl label pod labeled-pod app=frontend`
- List pods with a specific label:
 - `kubectl get pods -l app=frontend`
- Remove a label:
 - `kubectl label pod labeled-pod app-`

Taints and Tolerations in Kubernetes

Taints and Tolerations are mechanisms in Kubernetes to control which pods can run on which nodes. They ensure that pods are not scheduled onto inappropriate nodes.



1. Taints

- **Definition:** A taint is applied to a node to indicate that it should not accept any pod unless the pod explicitly tolerates the taint.
- **Syntax:**
 - `kubectl taint nodes <node-name> <key>=<value>:<effect>`
- **Effects:**

- NoSchedule: Pods that do not tolerate the taint will not be scheduled on the node.
- PreferNoSchedule: The scheduler avoids placing pods that do not tolerate the taint, but it's not guaranteed.
- NoExecute: Existing pods that do not tolerate the taint are evicted from the node.
- Example:

```
kubectl taint nodes node1 key1=value1:NoSchedule
```

This prevents pods that do not tolerate key1=value1 from being scheduled on node1.


2. Tolerations

- **Definition:** A toleration is applied to pods to let them be scheduled on nodes with matching taints.
- **Syntax** (in a Pod specification):



```
1 tolerations:  
2 - key: "key1"  
3   operator: "Equal"  
4   value: "value1"  
5   effect: "NoSchedule"
```

-
- **Fields:**
 - key: The taint key to tolerate.
 - operator: Determines the matching condition (Equal or Exists).
 - value: The taint value to tolerate.
 - effect: Must match the node taint's effect (NoSchedule, PreferNoSchedule, NoExecute).
- **Example:**



```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: example-pod
5  spec:
6    tolerations:
7    - key: "key1"
8      operator: "Equal"
9      value: "value1"
10     effect: "NoSchedule"
```

How They Work Together

1. **Taint on Node:** Restricts pods from being scheduled unless they have a matching toleration.
2. **Toleration in Pod:** Allows the pod to bypass the taint on a node and be scheduled.

Use Cases

- Dedicated nodes for specific workloads.
- Isolating nodes with special hardware (e.g., GPUs).
- Maintenance or decommissioning of nodes (using NoExecute).

Author
Subrahmanyam Penujuli

