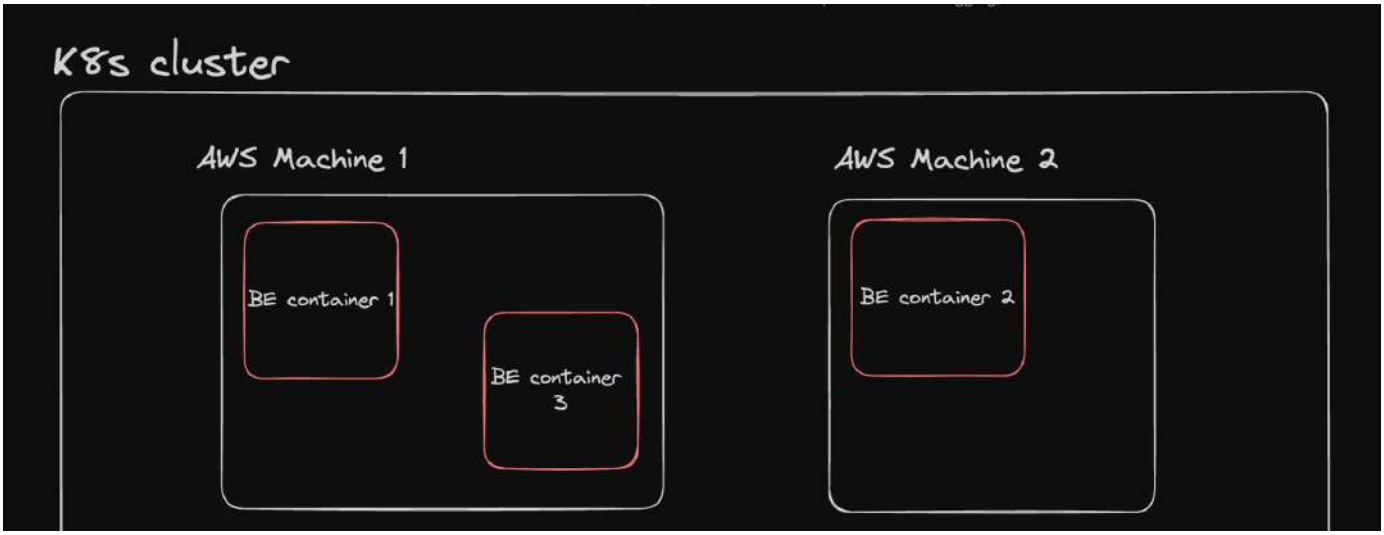
What is kubernetes

> 💡 Docker is a pre-requisite before you proceed to understand kubernetes

Kubernetes (popularly known as k8s) is a `container orchestration engine` , which as the name suggests lets you create, delete, and update `containers`

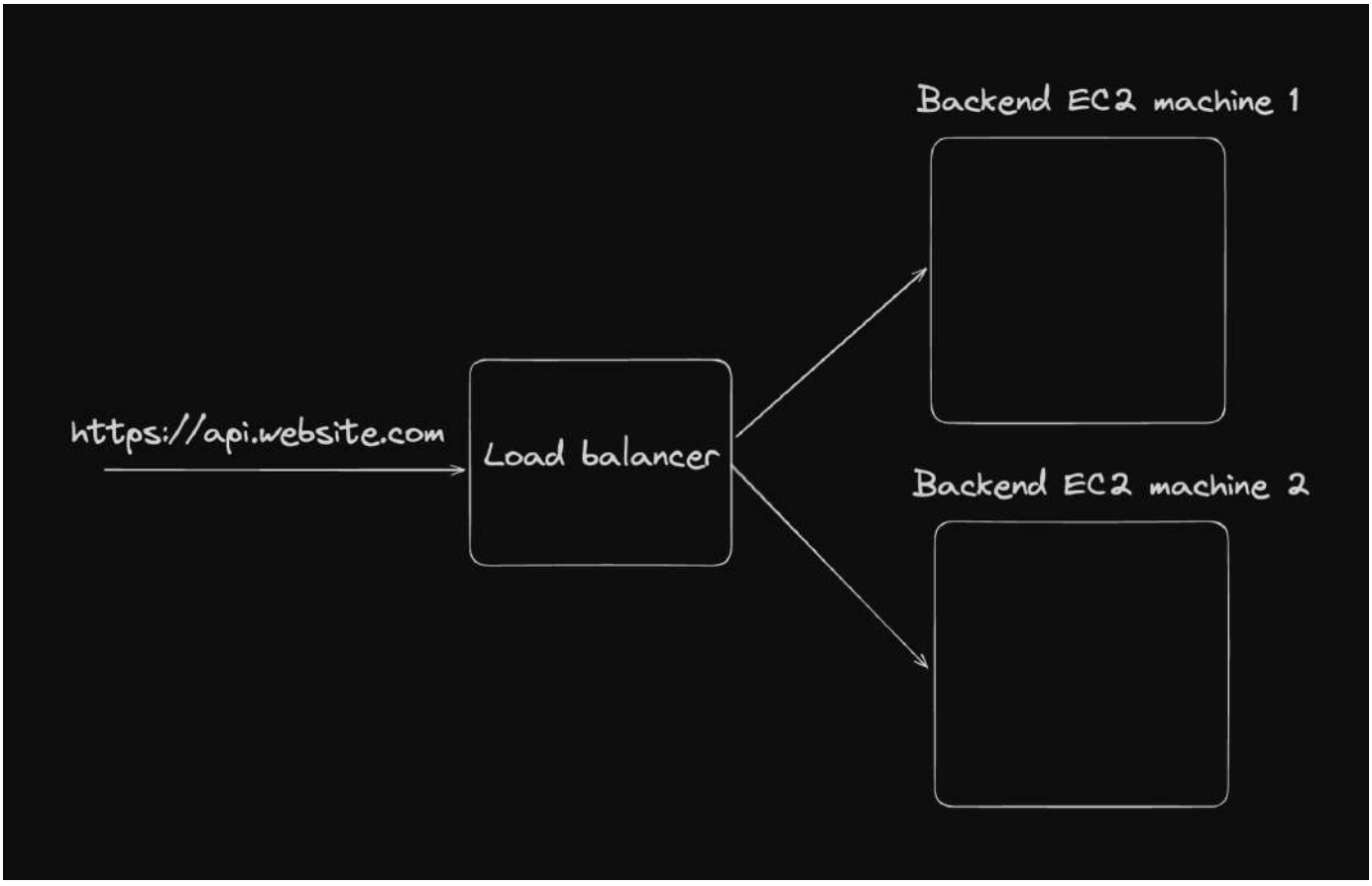
This is useful when

1. You have your docker images in the docker registry and want to deploy it in a `cloud native` fashion
2. You want to `not worry about` patching, crashes. You want the system to `auto heal`
3. You want to autoscale with some simple constructs
4. You want to observe your complete system in a simple dashboard

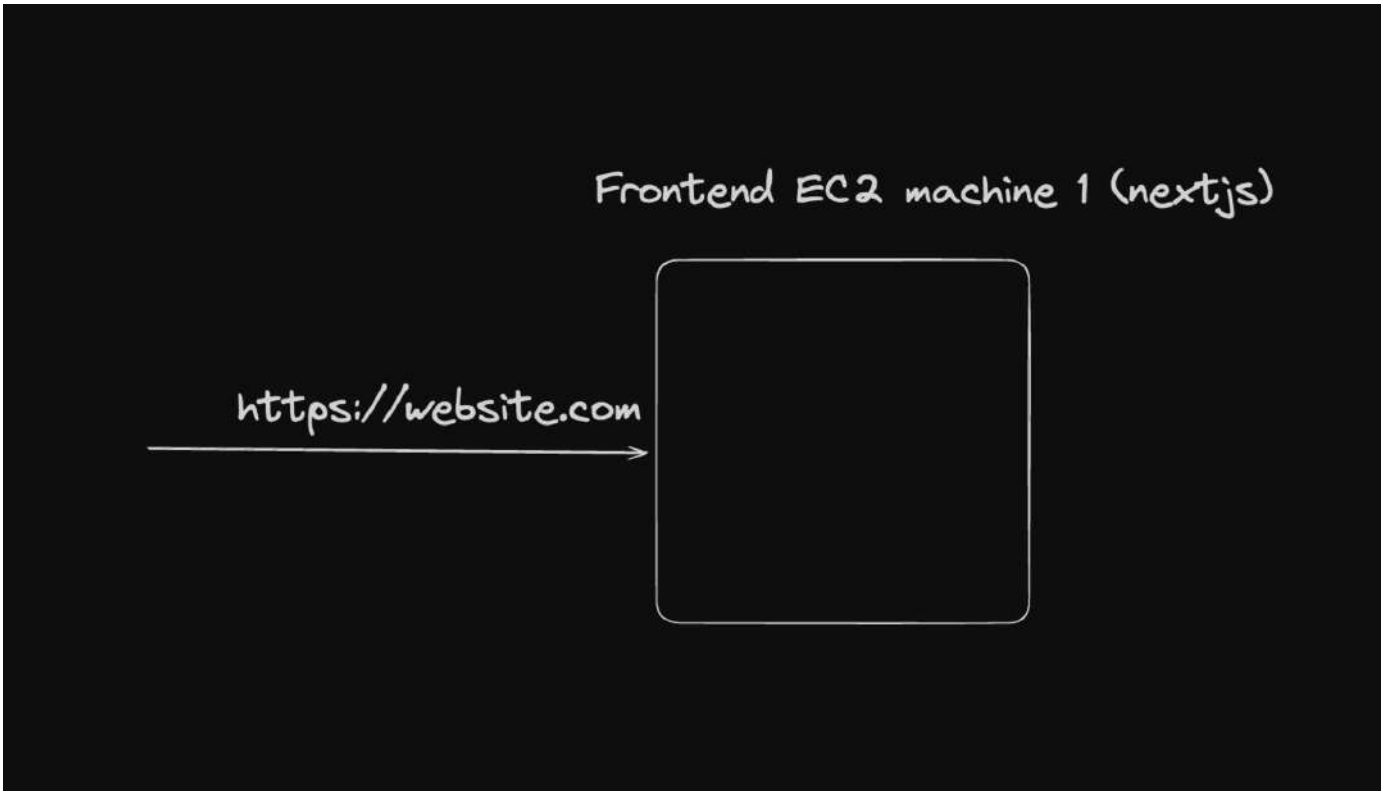


Before kubernetes

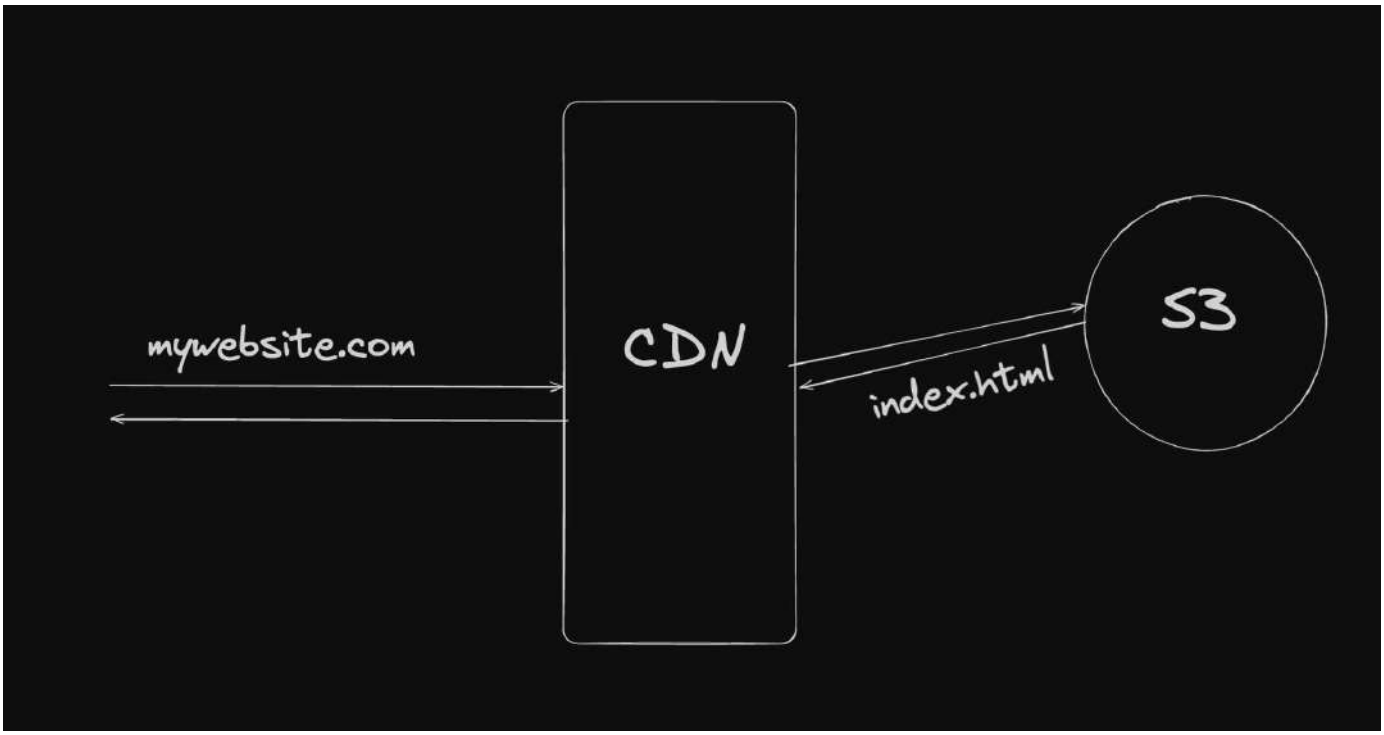
Backend



Frontend (Nextjs)

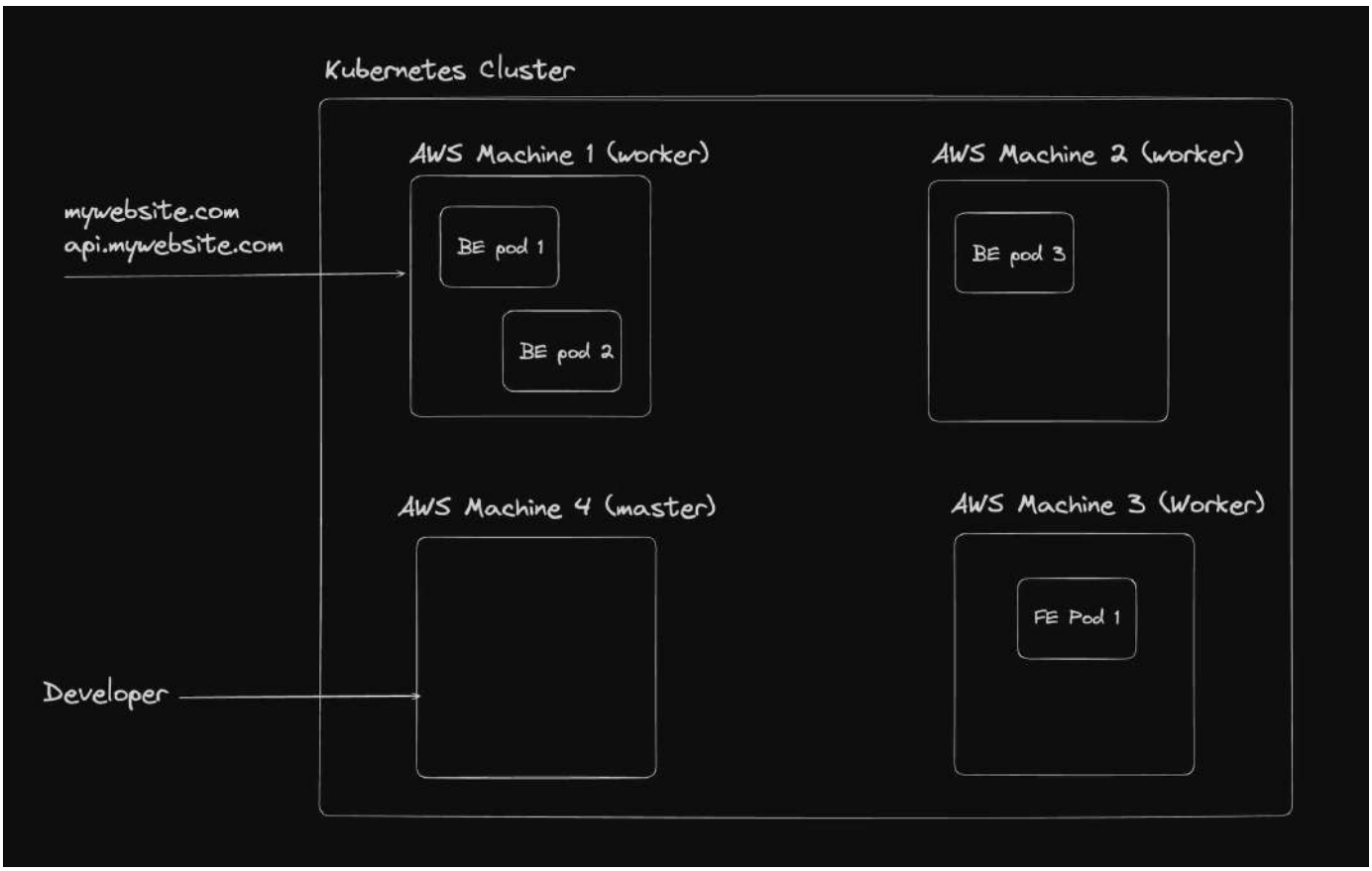


Frontend (React)



After kubernetes

Your frontend, backend are all `pods` in your `kubernetes cluster`



Jargon

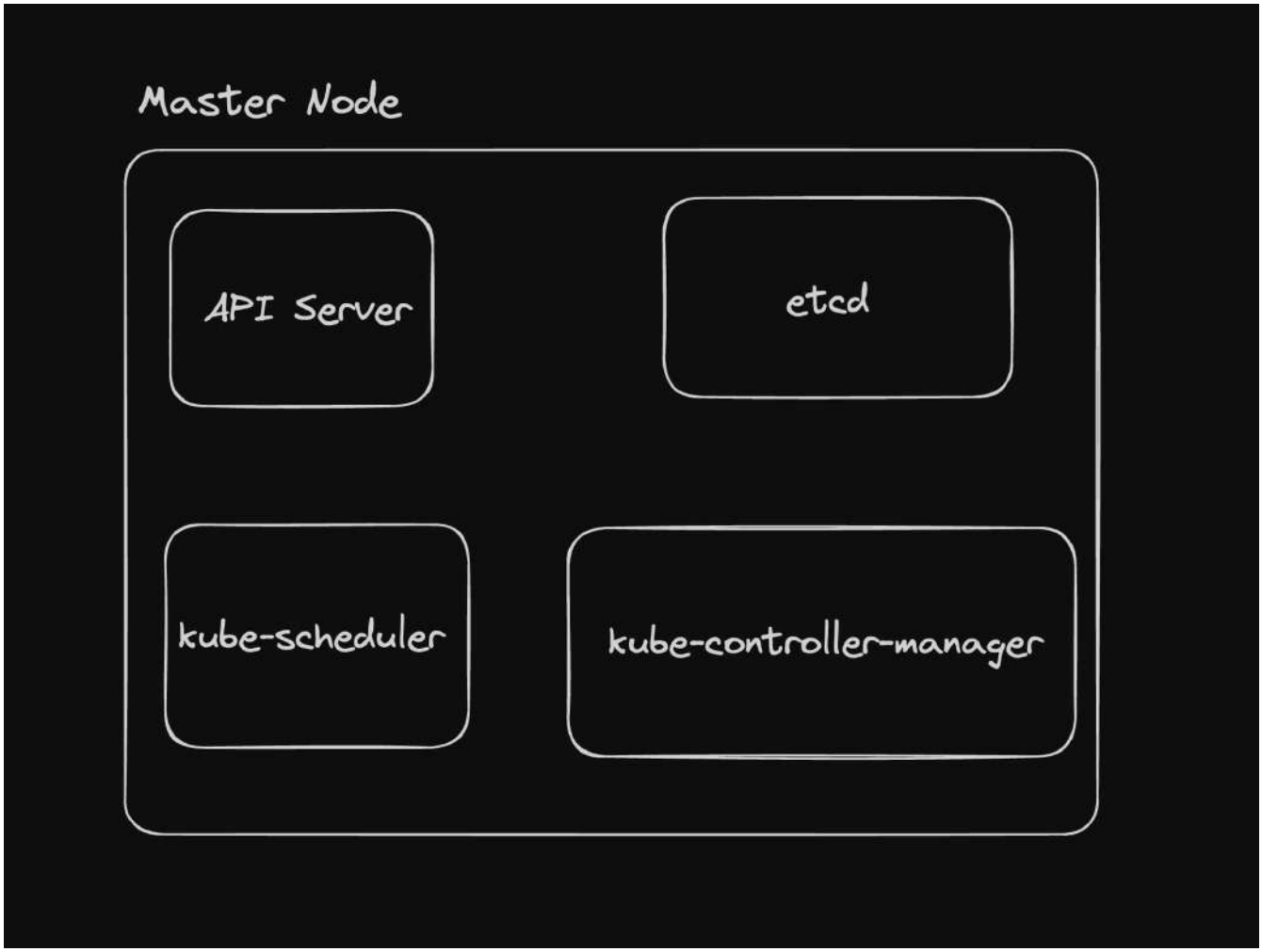
Ref - https://kubernetes.io/docs/concepts/overview/components/

Nodes

In kubernetes, you can create and connect various machines together, all of which are running `kubernetes` . Every machine here is known as a `node`

There are two types of nodes

▼ Master Node (Control pane) - The node that takes care of deploying the containers/healing them/listening to the developer to understand what to deploy



▼ API Server

1. **Handling RESTful API Requests**: The API server processes and responds to RESTful API requests from various clients, including the `kubectl` command-line tool, other Kubernetes components, and external applications. These requests involve creating, reading, updating, and deleting Kubernetes resources such as pods, services, and deployments
2. **Authentication and Authorization**: The API server authenticates and authorizes all API requests. It ensures that only authenticated and authorized users or components can perform actions on the cluster. This involves validating user credentials and checking access control policies.
3. **Metrics and Health Checks**: The API server exposes metrics and health check endpoints that can be used for monitoring and diagnosing the health and performance of the control plane.
4. **Communication Hub**: The API server acts as the central communication hub for the Kubernetes control plane. Other components, such as the scheduler, controller manager, and kubelet, interact with the API server to retrieve or update the state of the cluster.

▼ etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data. Ref - https://etcd.io/docs/v3.5/quickstart/

▼ kube-scehduler

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on. Its responsible for pod placement and deciding which pod goes on which node.

▼ **kube-controller-manager**

Ref - https://kubernetes.io/docs/concepts/architecture/controller/

The `kube-controller-manager` is a component of the Kubernetes control plane that runs a set of controllers. Each controller is responsible for managing a specific aspect of the cluster's state.

There are many different types of controllers. Some examples of them are:

- Node controller: Responsible for noticing and responding when nodes go down.
- Deployment controller: Watches for newly created or updated deployments and manages the creation and updating of ReplicaSets based on the deployment specifications. It ensures that the desired state of the deployment is maintained by creating or scaling ReplicaSets as needed.
- **ReplicaSet Controller**: Watches for newly created or updated ReplicaSets and ensures that the desired number of pod replicas are running at any given time. It creates or deletes pods as necessary to maintain the specified number of replicas in the ReplicaSet's configuration.

▼ Worker Nodes - The nodes that actually run your Backend/frontend

▼ **kubelet -** An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod.

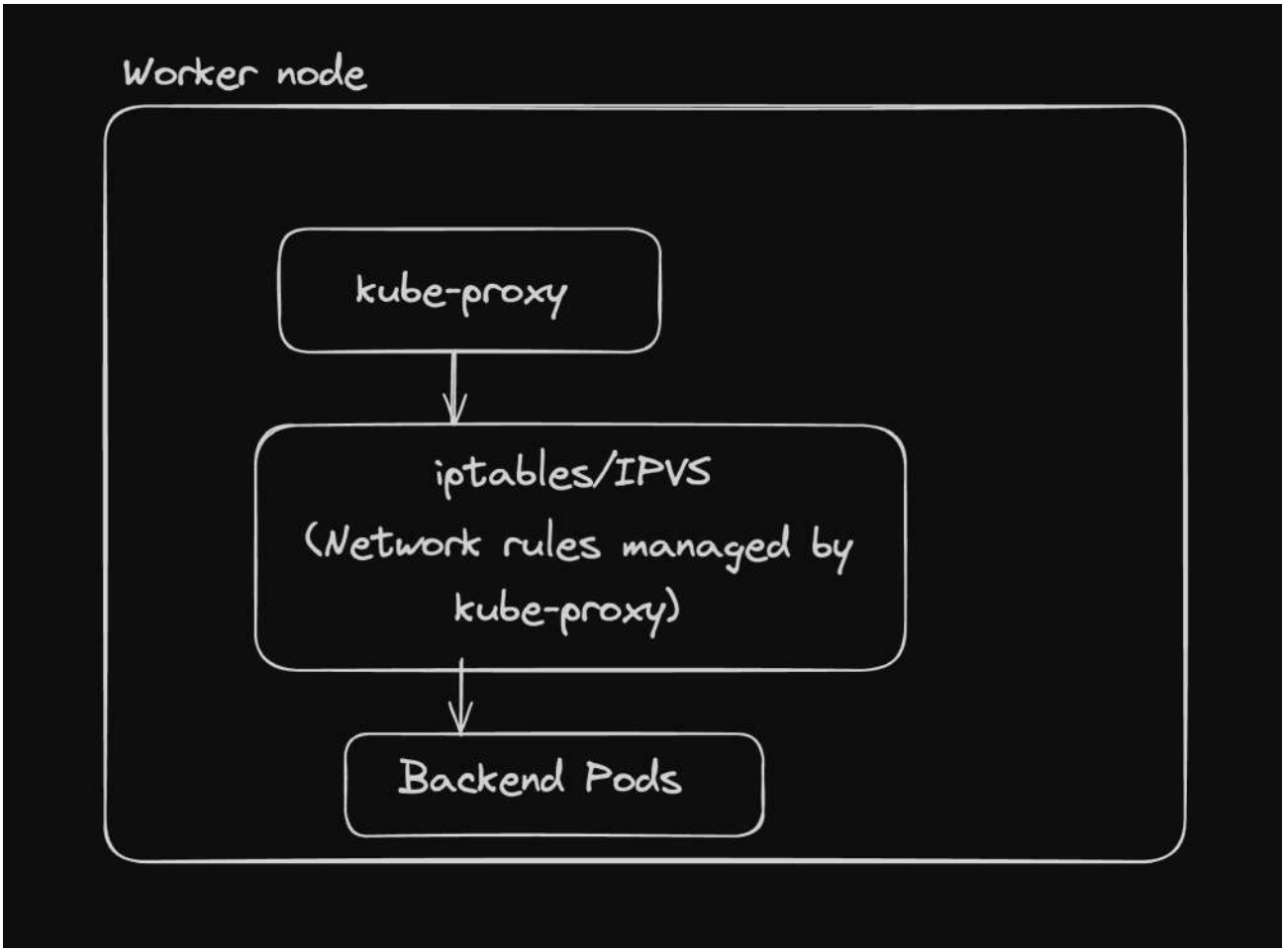
How the kubelet Control Loop Works

1. **Watch for PodSpecs**: The kubelet watches the API server for PodSpecs that are scheduled to run on its node. This includes both new pods that need to be started and existing pods that may need to be updated or terminated.
2. **Reconcile Desired State**: The kubelet compares the current state of the node (which pods are running, their statuses, etc.) with the desired state as defined by the PodSpecs.
3. **Take Action**: Based on this comparison, the kubelet takes actions to reconcile the actual state with the desired state:

- **Start Pods**: If there are new PodSpecs, the kubelet will pull the necessary container images, create the containers, and start the pods.
- **Monitor Health**: The kubelet performs health checks (liveness and readiness probes) on running containers. If a container fails a health check, the kubelet may restart it according to the pod's restart policy.
- **Update Pods**: If there are changes to the PodSpecs (e.g., configuration changes), the kubelet will update the running pods accordingly.
- **Stop Pods**: If a pod is no longer needed or should be terminated, the kubelet will stop and remove the containers.

4. **Report Status**: The kubelet periodically reports the status of the pods and the node back to the API server. This includes resource usage (CPU, memory, etc.) and the status of each container.

▼ **kube-proxy -** The `kube-proxy` is a network proxy that runs on each node in a Kubernetes cluster. It is responsible for you being able to talk to a pod



▼ **Container runtime** - In a Kubernetes worker node, the container runtime is the software responsible for running containers.

It interacts with the kubelet, which is the agent running on each node, to manage the lifecycle of containers as specified by Kubernetes pod specifications. The container runtime

ensures that the necessary containers are started, stopped, and managed correctly on the worker node.

Common Container Runtimes for Kubernetes

1. containerd
2. CRI-O
3. Docker

Kubernetes Container Runtime Interface (CRI)

The Container Runtime Interface (CRI) is a plugin interface that allows the kubelet to use a variety of container runtimes without needing to know the specifics of each runtime. This abstraction layer enables Kubernetes to support multiple container runtimes, providing flexibility and choice to users.

Cluster

A bunch of worker nodes + master nodes make up your `kubernetes cluster` . You can always add more / remove nodes from a cluster.

Images

A **Docker image** is a lightweight, standalone, and executable software package that includes everything needed to run a piece of software, including the code, runtime, libraries, environment variables, and configuration files. Images are built from a set of instructions defined in a file called a Dockerfile.

Eg - https://hub.docker.com/_/mongo

Containers

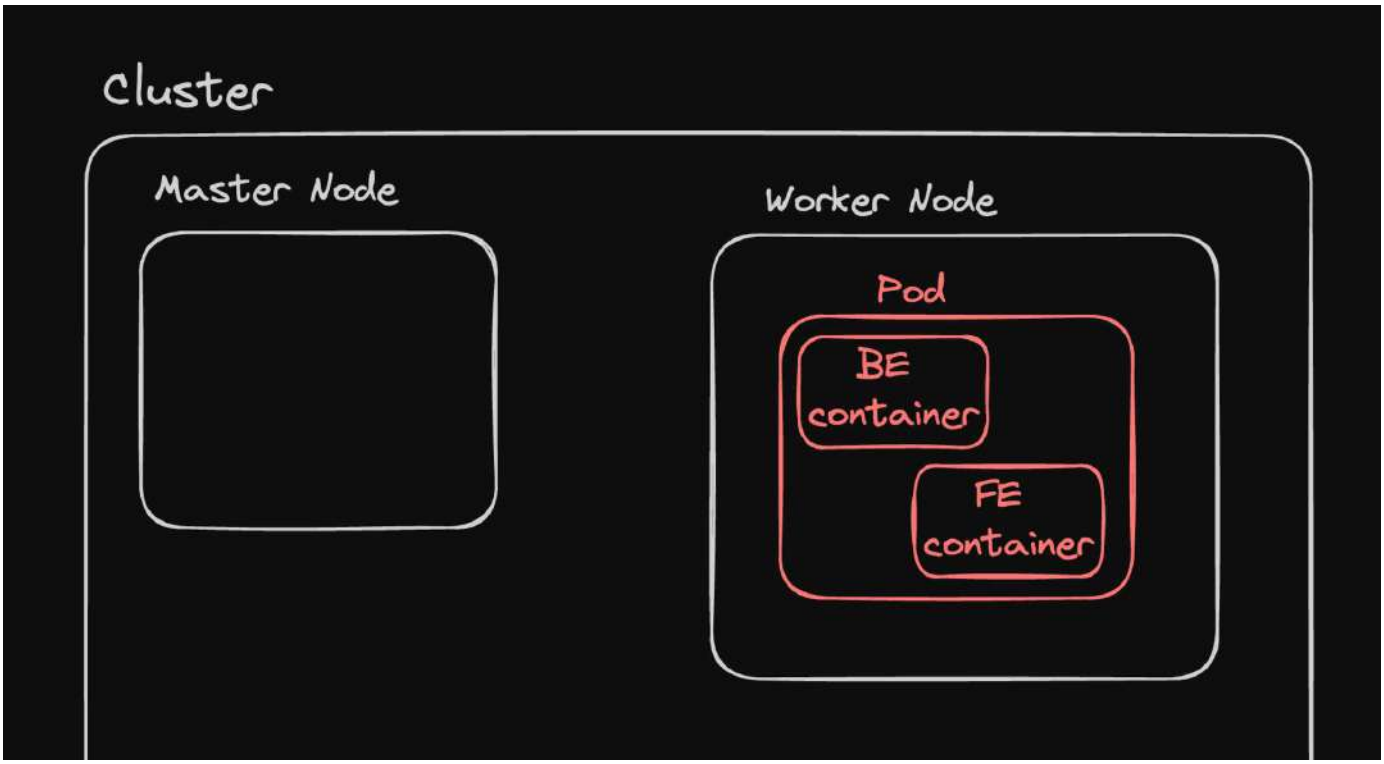
A container is an image in execution. For example if you run

```
docker run -p 5432:5432 -e POSTGRES_PASSWORD=mysecretpassword  -d postgres
```

Copy

Pods

A pod is the smallest and simplest unit in the Kubernetes object model that you can create or deploy



Creating a k8s cluster

Locally (Make sure you have docker)

1. minukube
2. kind - https://kind.sigs.k8s.io/docs/user/quick-start/

On cloud

1. GKE
2. AWS K8s
3. vultr

Using kind

- Install kind - https://kind.sigs.k8s.io/docs/user/quick-start/#installation

Single node setup

- Create a 1 node cluster

```
kind create cluster --name local
```
Copy

- Check the docker containers you have running

```
docker ps
```
Copy

- You will notice a single container running (control-pane)
- Delete the cluster

```
kind delete cluster -n local
```
Copy

Multi node setup

- Create a `clusters.yml` file

```
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
- role: worker
```
Copy

- Create the node setup

```
kind create cluster --config clusters.yml --name local
```
Copy

- Check docker containers

```
docker ps
```
Copy

```
→  week-26-prom-offline docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED         STATUS         PORTS                     NAMES
a3e949b85e91   kindest/node:v1.30.0   "/usr/local/bin/entr…"   2 minutes ago   Up 2 minutes   127.0.0.1:50949->6443/tcp   local-control-plane
49f7f1f3cd1d   kindest/node:v1.30.0   "/usr/local/bin/entr…"   2 minutes ago   Up 2 minutes                               local-worker2
ee0b793c157a   kindest/node:v1.30.0   "/usr/local/bin/entr…"   2 minutes ago   Up 2 minutes                               local-worker
```

Now you have a node cluster running locally

Using minikube

- Install minikube - https://minikube.sigs.k8s.io/docs/start/?arch=%2Fmacos%2Fx86-64%2Fstable%2Fbinary+download
- Start a k8s cluster locally

```
minikube start
```

Copy

- Run `docker ps` to see the single node setup

> 💡 A single node setup works but is not ideal. You don't want your control pane to run containers/act as a worker.

Kubernetes API

The master node (control pane) exposes an API that a developer can use to start pods.

Try the API

- Run `docker ps` to find where the control pane is running
- Try hitting various endpoints on the API server - https://127.0.0.1:50949/api/v1/namespaces/default/pods

kubectl

`kubectl` is a command-line tool for interacting with Kubernetes clusters. It provides a way to communicate with the Kubernetes API server and manage Kubernetes resources.

Install kubectl

https://kubernetes.io/docs/tasks/tools/#kubectl

Ensure kubectl works fine

```
kubectl get nodes
kubectl get pods
```

Copy

If you want to see the exact HTTP request that goes out to the API server, you can add `--v=8` flag

```
kubectl get nodes -v=8
```
Copy

Creating a Pod

There were 5 jargons we learnt about

1. Cluster
2. Nodes
3. Images
4. Containers
5. Pods

We have created a `cluster` of `3 nodes`

How can we deploy a `single container` from `an image` inside a `pod` ?

Finding a good image

Let's try to start this image locally - https://hub.docker.com/_/nginx

Starting using docker

```
docker run -p 3005:80 nginx
```
Copy

Try visiting localhost:3005

Starting a pod using k8s

- Start a pod

```
kubectl run nginx --image=nginx --port=80
```
Copy

- Check the status of the pod

```
kubectl get pods
```
Copy

- Check the logs

```
kubectl logs nginx
```
Copy

- Describe the pod to see more details

What our system looks like right now

Good questions to ask

1. How can I stop a pod?
2. How can I visit the pod? Which port is it available on?
3. How many pods can I start?

Stop the pod

Stop the pod by running

```
kubectl delete pod nginx
```

Copy

Check the current state of pods

```
kubectl get pods
```

Copy

Kubernetes manifest

A manifest defines the desired state for Kubernetes resources, such as Pods, Deployments, Services, etc., in a declarative manner.

Original command

```
kubectl run nginx --image=nginx --port=80
```

Copy

Manifest

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
    ports:
    - containerPort: 80
```

Copy

Breaking down the manifest

Applying the manifest

```
kubectl apply -f manifest.yml
```

Copy

Delete the pod

```
kubectl delete pod nginx
```
Copy

Checkpoint

We have created

1. Cluster
2. Nodes
3. Images
4. Containers
5. Pods

Deployment

A **Deployment** in Kubernetes is a higher-level abstraction that manages a set of Pods and provides declarative updates to them. It offers features like scaling, rolling updates, and rollback capabilities, making it easier to manage the lifecycle of applications.

- **Pod**: A Pod is the smallest and simplest Kubernetes object. It represents a single instance of a running process in your cluster, typically containing one or more containers.
- **Deployment**: A Deployment is a higher-level controller that manages a set of identical Pods. It ensures the desired number of Pods are running and provides declarative updates to the Pods it manages.

Key Differences Between Deployment and Pod:

1. **Abstraction Level**:
    - **Pod**: A Pod is the smallest and simplest Kubernetes object. It represents a single instance of a running process in your cluster, typically containing one or more containers.
    - **Deployment**: A Deployment is a higher-level controller that manages a set of identical Pods. It ensures the desired number of Pods are running and provides declarative updates to the Pods it manages.
2. **Management**:
    - **Pod**: They are ephemeral, meaning they can be created and destroyed frequently.

- **Deployment**: Deployments manage Pods by ensuring the specified number of replicas are running at any given time. If a Pod fails, the Deployment controller replaces it automatically.

3. **Updates**:
    - **Pod**: Directly updating a Pod requires manual intervention and can lead to downtime.
    - **Deployment**: Supports rolling updates, allowing you to update the Pod template (e.g., new container image) and roll out changes gradually. If something goes wrong, you can roll back to a previous version.
4. **Scaling**:
    - **Pod**: Scaling Pods manually involves creating or deleting individual Pods.
    - **Deployment**: Allows easy scaling by specifying the desired number of replicas. The Deployment controller adjusts the number of Pods automatically.
5. **Self-Healing**:
    - **Pod**: If a Pod crashes, it needs to be restarted manually unless managed by a higher-level controller like a Deployment.
    - **Deployment**: Automatically replaces failed Pods, ensuring the desired state is maintained.

Replicaset

A ReplicaSet in Kubernetes is a controller that ensures a specified number of pod replicas are running at any given time. It is used to maintain a stable set of replica Pods running in the cluster, even if some Pods fail or are deleted.

When you create a deployment, you mention the amount of `replicas` you want for this specific pod to run. The deployment then creates a new `ReplicaSet` that is responsible for creating X number of pods.

Series of events

User creates a `deployment` which creates a `replicaset` which creates `pods`

If `pods` go down, `replicaset controller` ensures to bring them back up

Series of events

When you run the following command, a bunch of things happen

```
kubectl create deployment nginx-deployment --image=nginx --port=80 --replicas=3
```

Copy

Step-by-Step Breakdown:

1. **Command Execution**:
    - You execute the command on a machine with `kubectl` installed and configured to interact with your Kubernetes cluster.

2. **API Request**:
    - `kubectl` sends a request to the Kubernetes API server to create a Deployment resource with the specified parameters.
3. **API Server Processing**:
    - The API server receives the request, validates it, and then processes it. If the request is valid, the API server updates the desired state of the cluster stored in etcd. The desired state now includes the new Deployment resource.
4. **Storage in etcd**:
    - The Deployment definition is stored in etcd, the distributed key-value store used by Kubernetes to store all its configuration data and cluster state. etcd is the source of truth for the cluster's desired state.
5. **Deployment Controller Monitoring**:
    - The Deployment controller, which is part of the `kube-controller-manager`, continuously watches the API server for changes to Deployments. It detects the new Deployment you created.
6. **ReplicaSet Creation**:
    - The Deployment controller creates a ReplicaSet based on the Deployment's specification. The ReplicaSet is responsible for maintaining a stable set of replica Pods running at any given time.
7. **Pod Creation**:
    - The ReplicaSet controller (another part of the `kube-controller-manager`) ensures that the desired number of Pods (in this case, 3) are created and running. It sends requests to the API server to create these Pods.
8. **Scheduler Assignment**:
    - The Kubernetes scheduler watches for new Pods that are in the "Pending" state. It assigns these Pods to suitable nodes in the cluster based on available resources and scheduling policies.
9. **Node and Kubelet**:
    - The kubelet on the selected nodes receives the Pod specifications from the API server. It then pulls the necessary container images (nginx in this case) and starts the containers.

▼ Hierarchical Relationship

1. **Deployment**:

- **High-Level Manager**: A Deployment is a higher-level controller that manages the entire lifecycle of an application, including updates, scaling, and rollbacks.
- **Creates and Manages ReplicaSets**: When you create or update a Deployment, it creates or updates ReplicaSets to reflect the desired state of your application.
- **Handles Rolling Updates and Rollbacks**: Deployments handle the complexity of updating applications by managing the creation of new ReplicaSets and scaling down old ones.

2. **ReplicaSet**:

- **Mid-Level Manager**: A ReplicaSet ensures that a specified number of identical Pods are running at any given time.
- **Maintains Desired State of Pods**: It creates and deletes Pods as needed to maintain the desired number of replicas.
- **Label Selector**: Uses label selectors to identify and manage Pods.

3. **Pods**:

- **Lowest-Level Unit**: A Pod is the smallest and simplest Kubernetes object. It represents a single instance of a running process in your cluster and typically contains one or more containers.

💡 A good question to ask at this point is why do you need a `deployment` when a `replicaset` is good enough to bring up and heal pods?

Create a replicaset

Let's not worry about deployments, lets just create a replicaset that starts 3 pods

- Create `rs.yml`

```
Copy
apiVersion: apps/v1
kind: ReplicaSet
metadata:
```

```
  name: nginx-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

- Apply the manifest

```
kubectl apply -f rs.yml
```

- Get the rs details

```
kubectl get rs

NAME               DESIRED   CURRENT   READY   AGE
nginx-replicaset   3         3         3       23s
```

- Check the pods

```
kubectl get pods

NAME                     READY   STATUS    RESTARTS   AGE
nginx-replicaset-7zp2v   1/1     Running   0          35s
nginx-replicaset-q264f   1/1     Running   0          35s
nginx-replicaset-vj42z   1/1     Running   0          35s
```

- Try deleting a pod and check if it self heals

```
kubectl delete pod nginx-replicaset-7zp2v
kubectl get pods
```

- Try adding a pod with the `app=nginx`

```
kubectl run nginx-pod --image=nginx --labels="app=nginx
```

Copy

- Ensure it gets terminated immedietely because the `rs` already has 3 pods
- Delete the replicaset

```
kubectl delete rs nginx-deployment-576c6b7b6
```

Copy

> 💡 Note the naming convention of the pods. The pods are named after the `replicaset` followed by a unique id (for eg nginx-replicaset-vj42z)

Create a deployment

Lets create a deployment that starts 3 pods

- Create deployment.yml

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

Copy

```
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

- Apply the deployment

```
kubectl apply -f deployment.yml
```

- Get the deployment

```
kubectl get deployment

NAME               READY   UP-TO-DATE   AVAILABLE   AGE
nginx-deployment   3/3     3            3           18s
```

- Get the rs

```
kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-576c6b7b6    3         3         3       34s
```

- Get the pod

```
kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
nginx-deployment-576c6b7b6-b6kgk    1/1     Running   0          46s
nginx-deployment-576c6b7b6-m8ttl    1/1     Running   0          46s
nginx-deployment-576c6b7b6-n9cx4    1/1     Running   0          46s
```

- Try deleting a pod

```
kubectl delete pod nginx-deployment-576c6b7b6-b6kgk
```

- Ensure the pods are still up

```
kubectl get pods
```
Copy

Why do you need deployment?

If all that a `deployment` does is create a `replicaset` , why cant we just create `rs` ?

Experiment

Update the `image` to be `nginx2` (an image that doesnt exist)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx2:latest
        ports:
        - containerPort: 80
```
Copy

- Apply the new deployment

```
kubectl apply -f deployment.yml
```

Copy

- Check the new `rs` now

```
kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-576c6b7b6    3         3         3       14m
nginx-deployment-5fbd4799cb   1         1         0       10m
```

Copy

- Check the pods

```
kubectl get pods
NAME                                READY   STATUS             RESTARTS   AGE
nginx-deployment-576c6b7b6-9nlnq    1/1     Running            0          15m
nginx-deployment-576c6b7b6-m8ttl    1/1     Running            0          16m
nginx-deployment-576c6b7b6-n9cx4    1/1     Running            0          16m
nginx-deployment-5fbd4799cb-fmt4f   0/1     ImagePullBackOff   0          12m
```

Copy

Role of deployment

Deployment ensures that there is a smooth deployment, and if the new image fails for some reason, the old replicaset is maintained.

Even though the `rs` is what does `pod management` , `deployment` is what does `rs management`

Rollbacks

- Check the history of deployment

```
kubectl rollout history deployment/nginx-deployment
```

Copy

- Undo the last deployment

```
kubectl rollout undo deployment/nginx-deployment
```

Copy

Create a new deployment

- Replace the image to be `postgres`

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: postgres:latest
        ports:
        - containerPort: 80
```

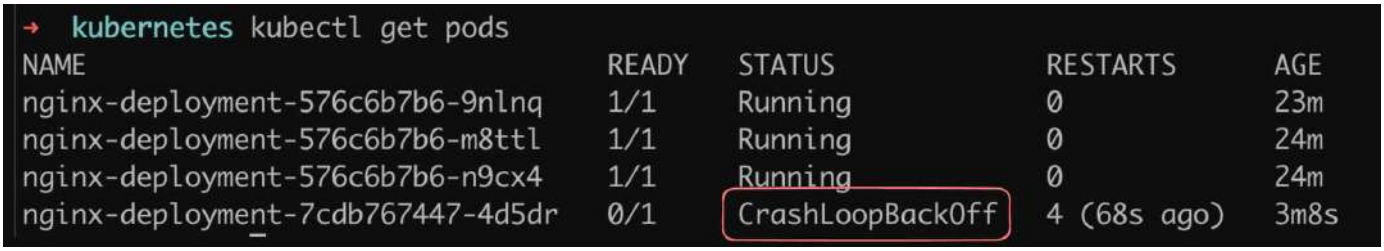
- Check the new set or `rs`

```
→  kubernetes kubectl get rs
NAME                          DESIRED   CURRENT   READY   AGE
nginx-deployment-576c6b7b6    3         3         3       21m
nginx-deployment-5fbd4799cb   0         0         0       17m
nginx-deployment-7cdb767447   1         1         0       4s
→  kubernetes kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
nginx-deployment-576c6b7b6-9nlnq    1/1     Running             0          20m
nginx-deployment-576c6b7b6-m8ttl    1/1     Running             0          21m
nginx-deployment-576c6b7b6-n9cx4    1/1     Running             0          21m
nginx-deployment-7cdb767447-4d5dr   0/1     ContainerCreating   0          7s
```

- Check the pods

```
→  kubernetes kubectl get pods
NAME                                READY   STATUS              RESTARTS   AGE
nginx-deployment-576c6b7b6-9nlnq    1/1     Running             0          20m
nginx-deployment-576c6b7b6-m8ttl    1/1     Running             0          21m
nginx-deployment-576c6b7b6-n9cx4    1/1     Running             0          21m
nginx-deployment-7cdb767447-4d5dr   0/1     ContainerCreating   0          7s
```

- Check pods after some time



- Check the logs

```
kubectl logs -f nginx-deployment-7cdb767447-4d5dr
Error: Database is uninitialized and superuser password is not specified.
       You must specify POSTGRES_PASSWORD to a non-empty value for the
       superuser. For example, "-e POSTGRES_PASSWORD=password" on "docker run".

       You may also use "POSTGRES_HOST_AUTH_METHOD=trust" to allow all
       connections without a password. This is *not* recommended.

       See PostgreSQL documentation about "trust":
       https://www.postgresql.org/docs/current/auth-trust.html
```

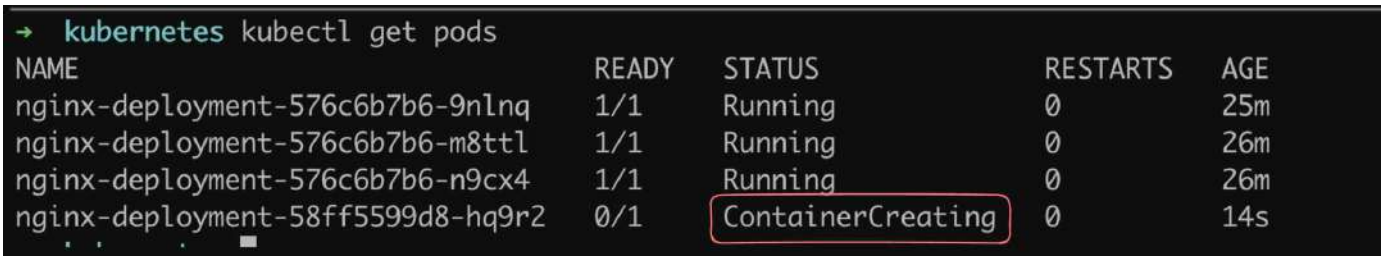
- Update the manifest to pass `POSTGRES_PASSWORD`

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: postgres:latest
        ports:
        - containerPort: 80
        env:
```

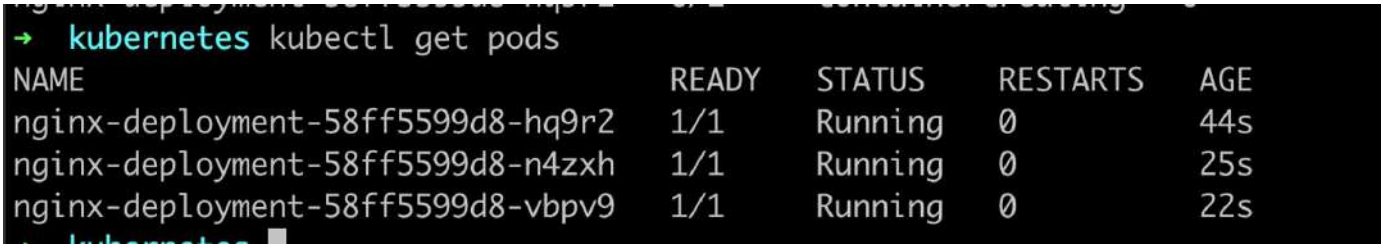
```
        - name: POSTGRES_PASSWORD
          value: "yourpassword"
```

- Check pods now

```
kubectl get pods
```

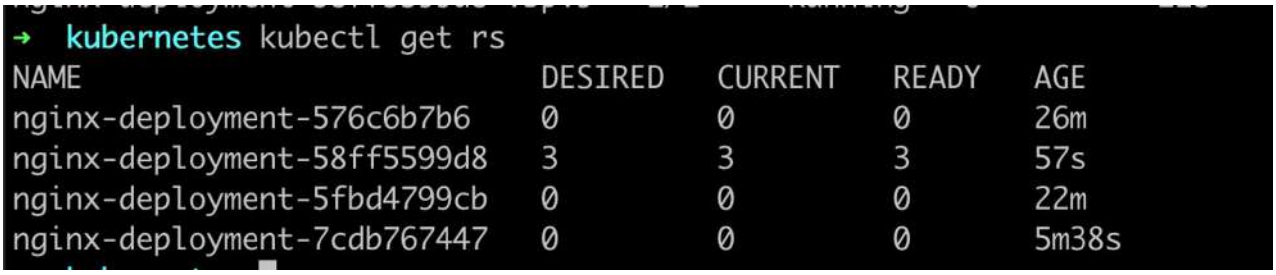


- Try after some time



Postgres is running correctly

- Check the rs



How to expose the app?

Let's delete all resources and restart a `deployment` for `nginx` with 3 replicas

```
apiVersion: apps/v1
kind: Deployment
metadata:
```

```
    name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
        ports:
        - containerPort: 80
```

- Apply the configuration

```
kubectl apply -f deployment.yml
```

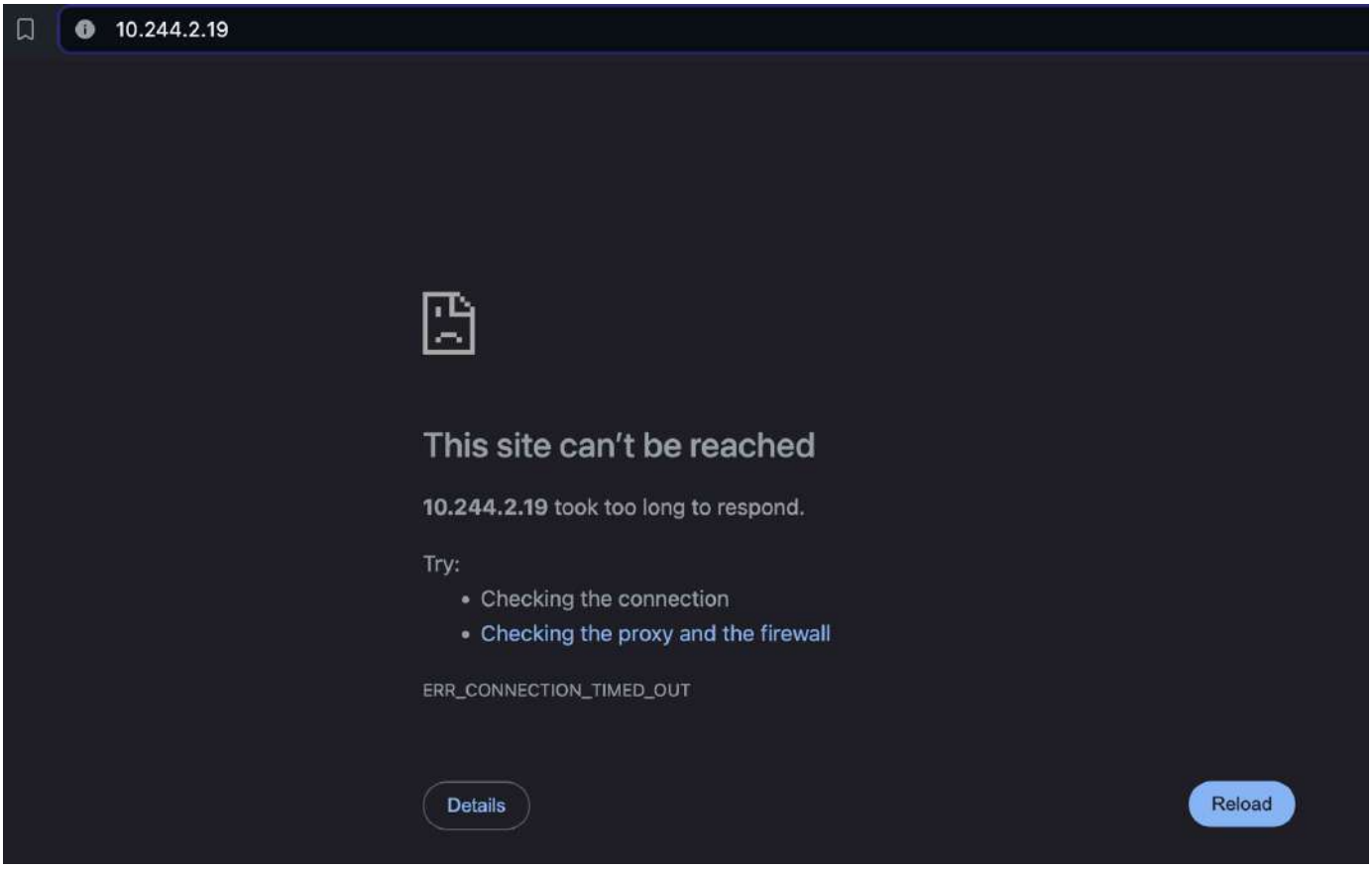
Copy

- Get all pods

```
kubectl get pods -owide
NAME                                READY   STATUS    RESTARTS   AGE     IP
nginx-deployment-576c6b7b6-7jrn5    1/1     Running   0          2m19s   10.244.
nginx-deployment-576c6b7b6-88fkh    1/1     Running   0          2m22s   10.244.
nginx-deployment-576c6b7b6-zf8ff    1/1     Running   0          2m25s   10.244.
```

Copy

- The IPs that you see are `private IPs` . You wont be able to access the app on it



Services

In Kubernetes, a "Service" is an abstraction that defines a logical set of Pods and a policy by which to access them. Kubernetes Services provide a way to expose applications running on a set of Pods as network services. Here are the key points about Services in Kubernetes:

▼ Key concepts

1. **Pod Selector**: Services use labels to select the Pods they target. A label selector identifies a set of Pods based on their labels.
2. **Service Types**:
    - **ClusterIP**: Exposes the Service on an internal IP in the cluster. This is the default ServiceType. The Service is only accessible within the cluster.
    - **NodePort**: Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You can contact the NodePort Service, from outside the cluster, by requesting `<NodeIP>:<NodePort>` .
    - **LoadBalancer**: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.

3. **Endpoints**: These are automatically created and updated by Kubernetes when the Pods selected by a Service's selector change.

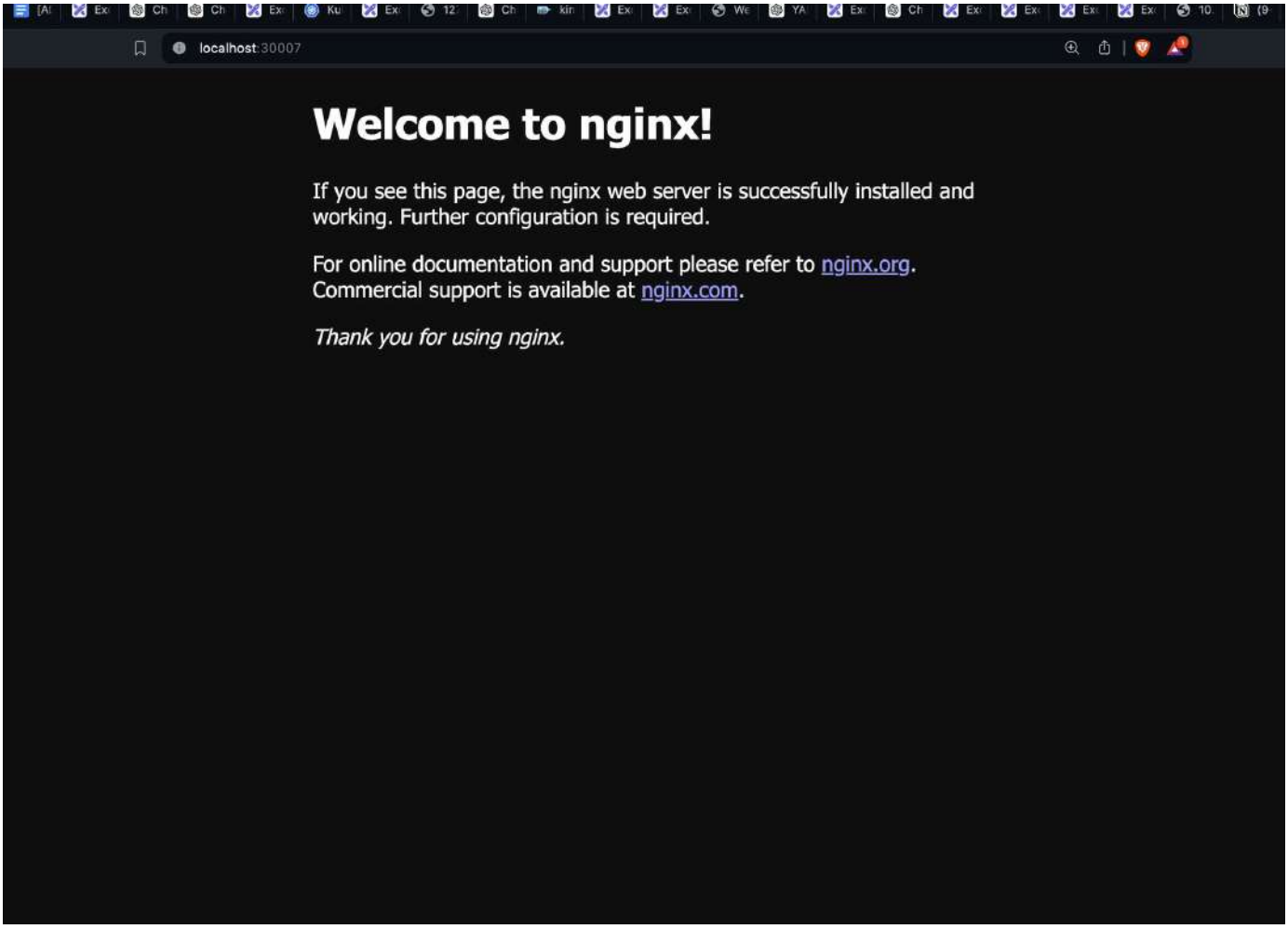
- Create service.yml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30007  # This port can be any valid port within the NodePort ra
  type: NodePort
```

- Restart the cluster with a few extra ports exposed (create kind.yml)

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 30007
    hostPort: 30007
- role: worker
- role: worker
```

- kind create cluster --config kind.yml
- Re apply the deployment and the service
- Visit `localhost:30007`



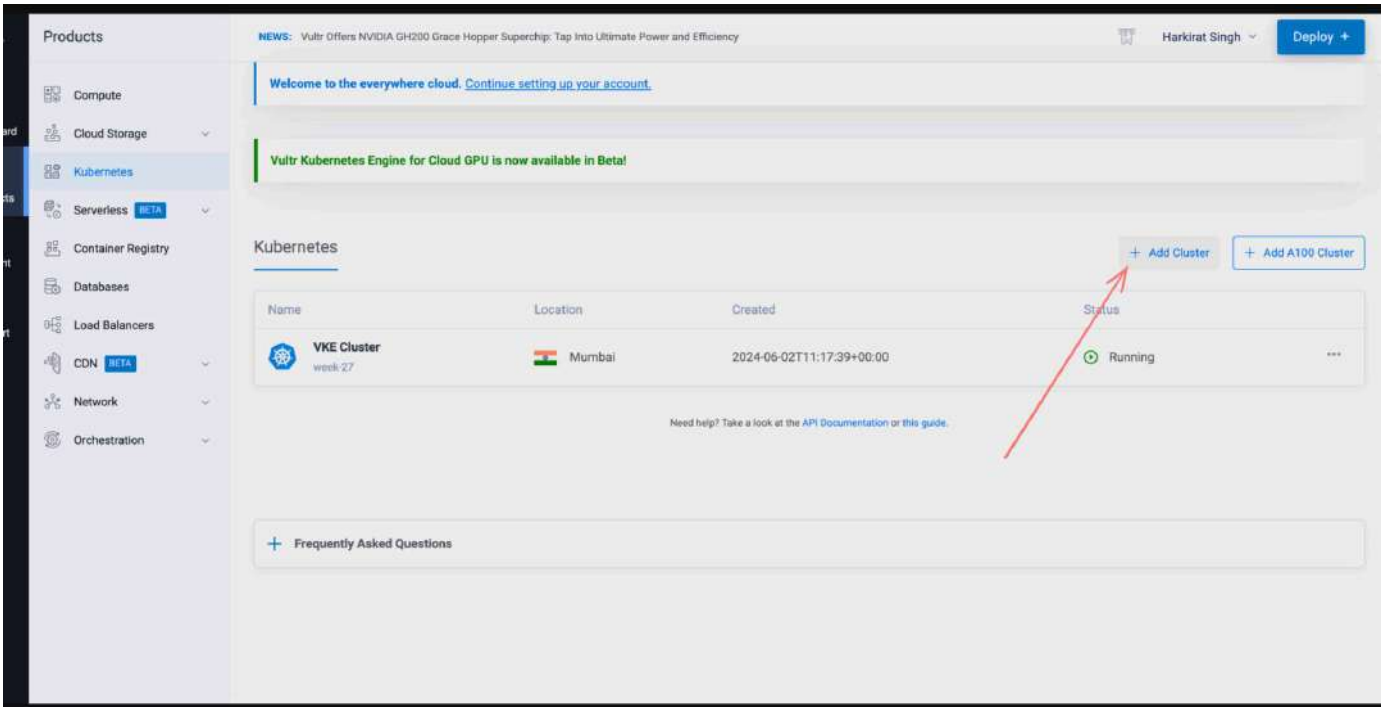
Types of services

1. ClusterIP

Loadbalancer service

In Kubernetes, a LoadBalancer service type is a way to expose a service to external clients. When you create a Service of type LoadBalancer, Kubernetes will automatically provision an `external` load balancer from your cloud provider (e.g., AWS, Google Cloud, Azure) to route traffic to your Kubernetes service

Creating a kubernetes cluster in vultr



- Create deployment.yml

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

- Apply deployment

```
kubectl apply -f deployment.yml
```
Copy

- Create `service-lb.yml`

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

- Apply the service

Copy

Series of events

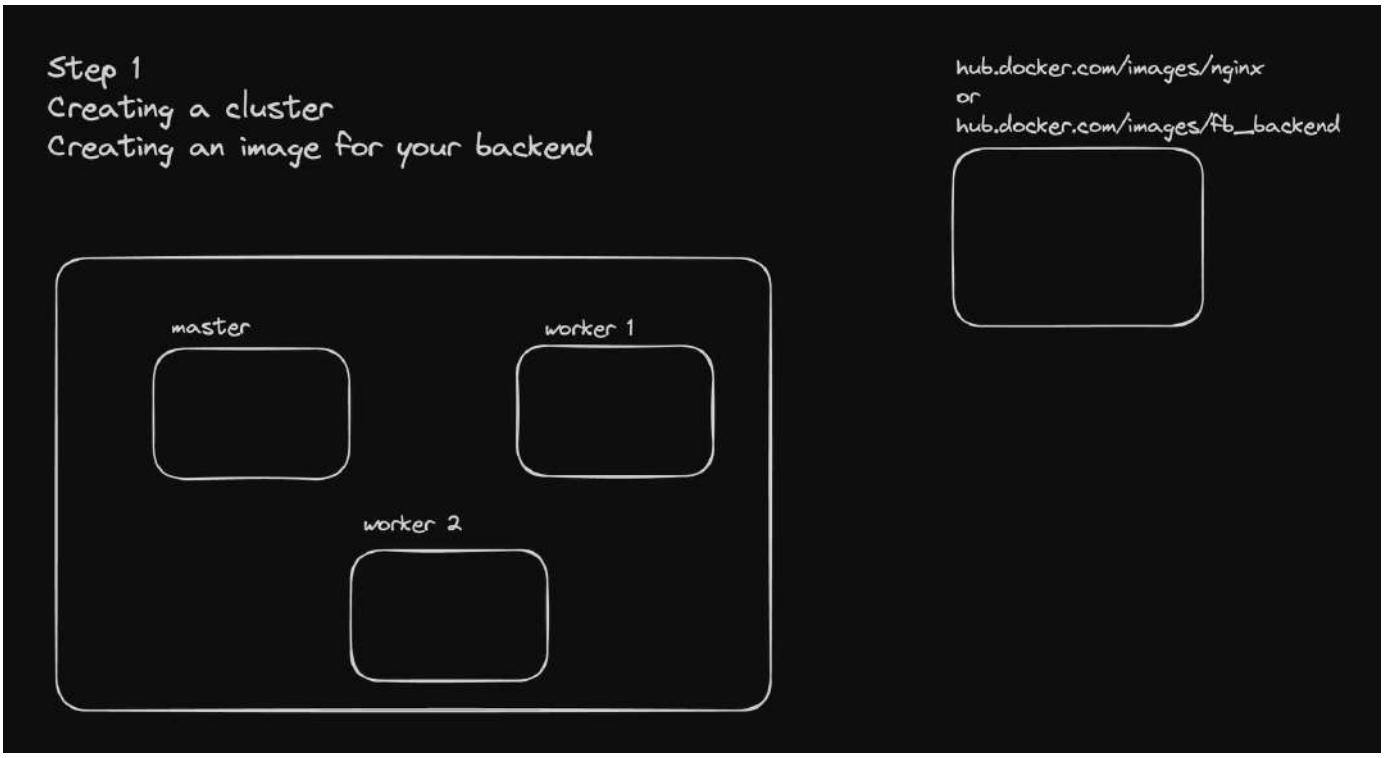
Step 1 - Create your cluster

- Create `kind.yml`

```yaml
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  extraPortMappings:
  - containerPort: 30007
    hostPort: 30007
- role: worker
  extraPortMappings:
  - containerPort: 30007
    hostPort: 30008
- role: worker
```
Copy

- Create cluster

```
kind create cluster --config kind.yml --name local
```
Copy



Step 2 - Deploy your pod

- Create `deployment.yml`

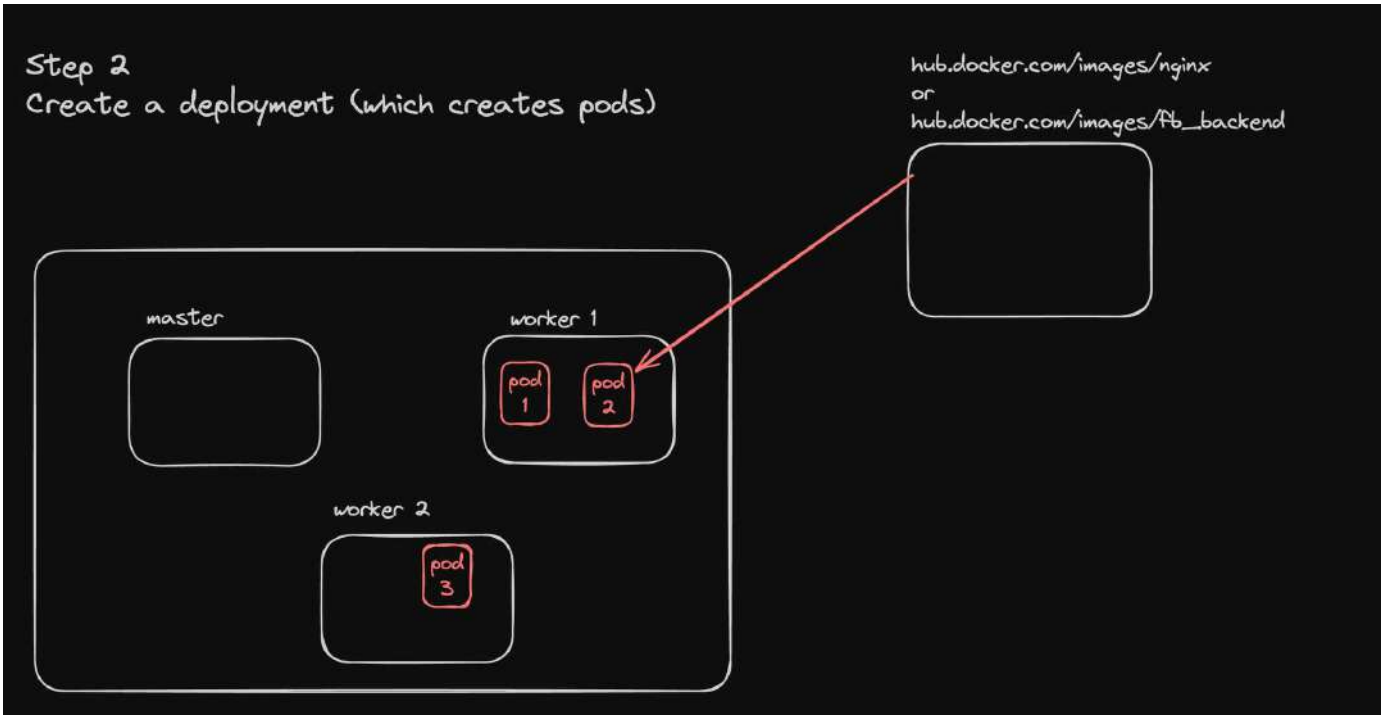
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:latest
```
Copy

```
        ports:
        - containerPort: 80
```

- Apply the deployment

```
kubectl apply -f deployment.yml
```

Copy



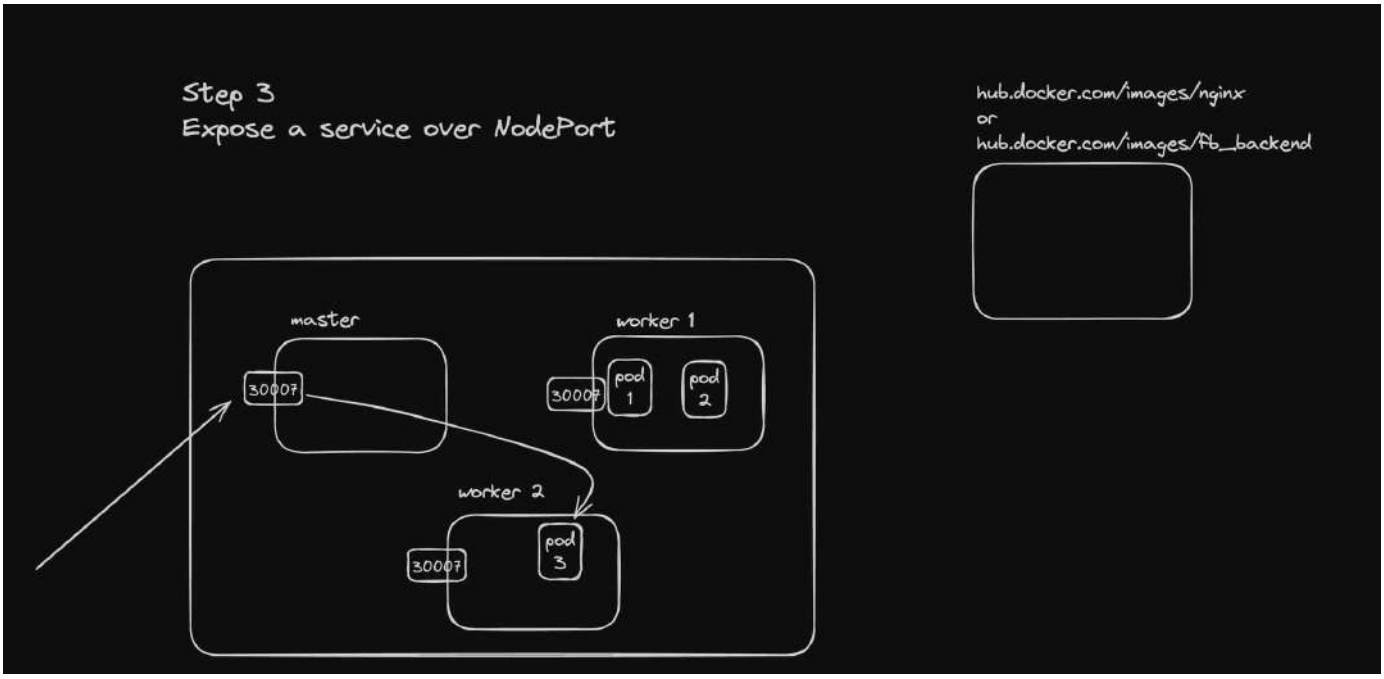
Step 3 - Expose your app over a NodePort

- Create service.yml

Copy

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
      nodePort: 30007  # This port can be any valid port within the NodePort ra
  type: NodePort
```

- kubectl apply -f service.yml



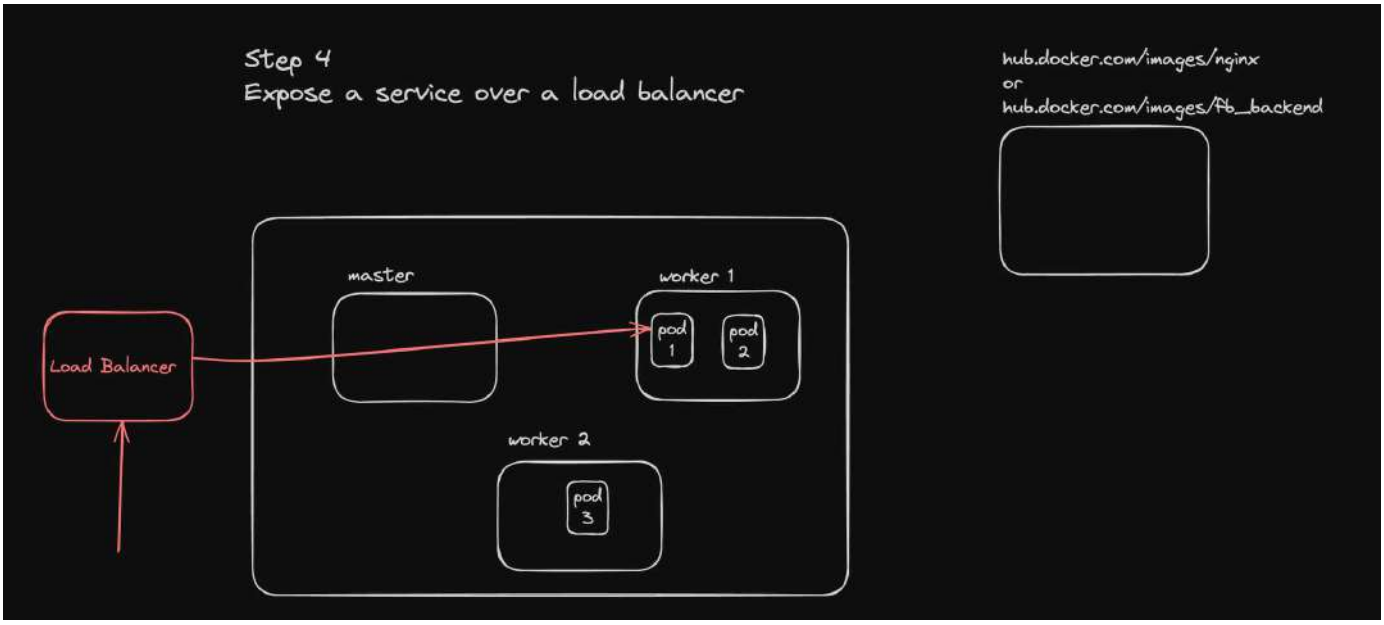
Step 4 - Expose it over a LoadBalancer

- Create a load balancer service (service-lb.yml)

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

- Apply the configuration

```
kubectl apply service-lb.yml
```



Check the cloud dashboard