What we'll be doing today

What we've done already

- 1. Clusters
- 2. Nodes
- 3. Pods
- 4. Deployment
- 5. Replicasets

What we're doing today

- 1. Namespaces
- 2. Ingress
- 3. Ingress Controller
 - 1. nginx
 - 2. traefik
- 4. ConfigMaps
- 5. Secrets

What we're doing tomorrow

- 1. Cert management
- 2. Volumes and Persistent volumes
- 3. Resource management

Offline video next week

- 1. HPA Horizontal Pod Autoscaling
- 2. Node autoscaling
- 3. Labs to add k8s a real codebase

Recapping what we've done

Ref - https://projects.100xdevs.com/tracks/kubernetes-1/Kubernetes-Part-1-1

Quick recap

▼ Cluster

A kubernetes cluster is a bunch of machines that work together to help you deploy your app

▼ Nodes

Each machine in your cluster is called a node

Nodes are of two types

- 1. Master node (control plane) Exposes an API that the developer can use to deploy pods
- 2. Worker node Actually run the pods

▼ Pode

The smallest execution unit inside a kubernetes cluseter. A pod can run one or more containers

▼ Replicasets

They let you create multiple pods (replicas).

It also takes care of bringing them back up if they ever go down/are killed

▼ Deployment

A deployment creates

▼ Services

Services let you expose your pods to other pods/over the internet

They are of three types

- 1. ClusterIP
- 2. NodePort
- 3. Loadbalancer Creates a loadbalancer outside the kubernetes cluster

Nodeport

Loadbalancer

Recapping how to run this locally

Creating a cluster

• Create a kind.yml file locally

```
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
```

• Run the cluster locally

```
kind create cluster --config kind.yml --name local2 Copy
```

• Run docker ps to confirm that the cluster is running

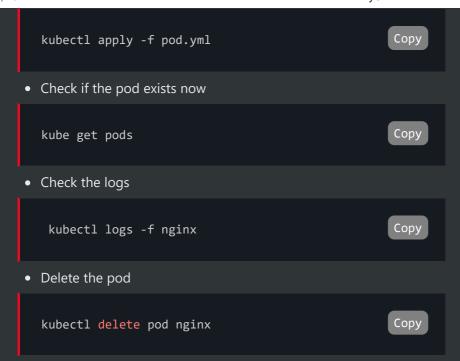
```
docker ps Copy
```

Creating a pod

• Create a pod manifest (pod.yml)

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

• Apply the pod manifest



Creating a replicaset

• Create the replicaset manifest

```
Сору
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 replicaset manifest

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

• Check the number of pods running now

```
kubectl get pods Copy
```

- Try deleting a pod, and ensure it gets restarted
- Delete the replicaset

```
kubectl delete rs nginx-replicaset Copy
```

Creating a Deployment

• Create a deployment manifest (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 the manifest

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

• Check the rs that exist now

```
kubectl get rs Copy
```

• Check the pods that exist now

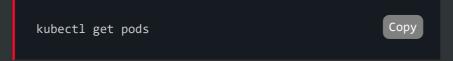
```
kubectl get pods Copy
```

• Try creating a new deployment with a wrong image name

```
apiVersion: apps/v1 Copy
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
```

• Ensure that the old pods are still running



Keep the deployment, it'll come in handy in the 4th slide

How to run this on a cloud provider

Go to a cloud provider like

- 1. AWS
- 2. GCP
- 3. Digital ocean
- 4. Vultr

and create a k8s cluster

- Create a cluster
- Download the credentials file and replace ~/.kube/config with it
- Create a deployment manifest

```
Сору
  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
• Create a deployment
                                                     Сору
  kubectl apply -f deployment.yml
```

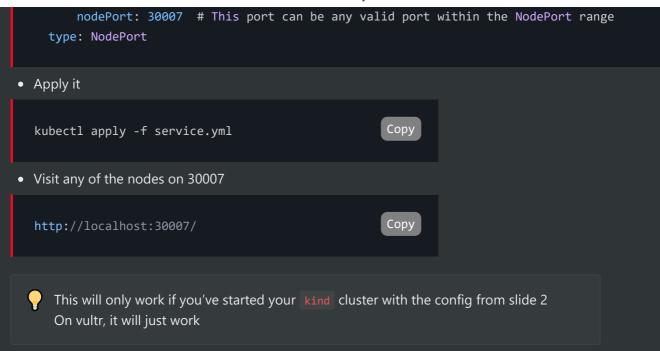
Services

Services let you actually expose your app over the internet.

Nodeport

• Create a Nodeport service (service.yml)

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



LoadBalancer (will only work on a cloud provider)

The LoadBalancer service type is designed to work with cloud providers to create an external load balancer that routes traffic to the service.

Сору

• Replace the type to be LoadBalancer

```
apiVersion: v1
kind: Service
metadata:
   name: nginx-service
spec:
   selector:
    app: nginx
   ports:
    - protocol: TCP
        port: 80
        targetPort: 80
        type: LoadBalancer
• Re-apply the config
```

- See the loadbalancer created on the dashboard
- Visit the balancer to see the website

kubectl apply -f service.yml

Downsides of services

Services are great, but they have some downsides -

Scaling to multiple apps

- 1. If you have three apps (frontend, backend, websocket server), you will have to create saparate services to route traffic to them. There is no way to do centralized traffic management (routing traffic from the same URL/Path-Based Routing)
- 2. There are also limits to how many load balancers you can create

Multiple certificates for every route

You can create certificates for your load balancers but you have to maintain them outside the cluster and create them manually

You also have to update them if they ever expire

No centralized logic to handle rate limitting to all services

Each load balancer can have its own set of rate limits, but you cant create a single rate limitter for all your services.

Trying it out

Here is a sample manifest that you can run to start two saparate deployments and attach them to two saparate LoadBalancer services

▼ Manifest

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: nginx-deployment
spec:
    replicas: 2
    selector:
        matchLabels:
        app: nginx
    template:
        metadata:
        labels:
        app: nginx
```

```
spec:
      containers:
      - name: nginx
        image: nginx:alpine
        ports:
        - containerPort: 80
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: my-apache-site
       image: httpd:2.4
        ports:
       - containerPort: 80
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  selector:
   app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
apiVersion: v1
kind: Service
metadata:
  name: apache-service
spec:
  selector:
    app: apache
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: LoadBalancer
```

Open the load balancers

kubectl apply -f manifest.yml

You will notice two load balancers created for your two services

Ingress and Ingress Controller

Ref - https://kubernetes.io/docs/concepts/services-networking/ingress/

An API object that manages external access to the services in a cluster, typically HTTP. Ingress may provide load balancing, SSL termination and name-based virtual hosting.



An Ingress does not expose arbitrary ports or protocols. Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.

Ingress controller

If you remember from last week, our control plane had a controller manager running.

Ref - https://projects.100xdevs.com/tracks/kubernetes-1/Kubernetes-Part-1-3

The kube-controller-manager runs a bunch of controllers like

- 1. Replicaset controller
- 2. Deployment controller

etc

If you want to add an ingress to your kubernetes cluster, you need to install an ingress controller manually. It doesn't come by default in k8s

Famous k8s ingress controllers

- The NGINX Ingress Controller for Kubernetes works with the NGINX webserver (as a proxy).
- HAProxy Ingress is an ingress controller for HAProxy.
- The Traefik Kubernetes Ingress provider is an ingress controller for the Traefik proxy.

Full list - https://kubernetes.io/docs/concepts/services-networking/ingress-controllers/

Namespaces

In Kubernetes, a namespace is a way to divide cluster resources between multiple users/teams. Namespaces are intended for use in environments with many users spread across multiple teams, or projects, or environments like development, staging, and production.

When you do

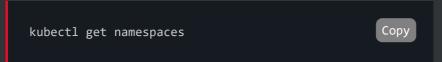


Creating a new namespace

• Create a new namespace



• Get all the namespaces



• Get all pods in the namespace



• Create the manifest for a deployment in the namespace

```
Сору
  apiVersion: apps/v1
  kind: Deployment
  metadata:
    name: nginx-deployment
    namespace: backend-team
  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

                                                      Сору
  kubectl apply -f deployment-ns.yml
• Get the deployments in the namespace
  kubectl get deployment -n backend-team
                                                      Сору

    Get the pods in the namespace

  kubectl get pods -n backend-team
                                                      Copy
• Set the default context to be the namespace
                                                                      Сору
  kubectl config set-context --current --namespace=backend-team
• Try seeing the pods now
  kubectl get pods
                                                       Copy
• Revert back the kubectl config
                                                                 Copy
  kubectl config set-context --current --namespace=default
```

Install the nginx ingress controller

Ref - https://docs.nginx.com/nginx-ingress-controller/installation/installing-nic/installation-with-manifests/

Using helm

• Install helm

Ref - https://helm.sh/

Installation - https://helm.sh/docs/intro/install/

• Add the ingress-nginx chart

helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx helm repo update helm install nginx-ingress ingress-nginx/ingress-nginx --namespace ingress-nginx --c

· Check if you have pods running in the

kubectl get pods -n ingress-nginx Copy

Default loadbalancer service

You will notice that if you use helm to install the nginx-ingress-controller, it creates a Loadbalancer service for you

kubectl get services --all-namespaces Copy

This routes all the traffic to an nginx pod

kubectl get pods -n ingress-nginx Copy

This means the first part of our ingress deployment is already created

Adding the routing to the ingress controller

Next up, we want to do the following -

• Get rid of all existing deployments in the default namespace

```
kubectl get deployments
kubectl delete deployment_name
```

• Get rid of all the services in the default namespace (dont delete the default kubernetes service, delete the old nginx and apache loadbalancer services)

```
kubectl get services
kubect
```

• Create a deployment and service definition for the nginx image/app (this is different from the nginx controller)

```
Copy
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
        ports:
        - containerPort: 80
apiVersion: v1
```

```
kind: Service
metadata:
 name: nginx-service
 namespace: default
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

• Create a deployment and service for the apache app

```
Сору
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
      - name: my-apache-site
        image: httpd:2.4
        ports:
        - containerPort: 80
apiVersion: v1
kind: Service
metadata:
  name: apache-service
  namespace: default
spec:
  selector:
   app: apache
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
```

Create the ingress resource

```
Сору
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-apps-ingress
  namespace: default
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
  ingressClassName: nginx
  rules:
  - host: your-domain.com
    http:
      paths:
      - path: /nginx
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              number: 80
      - path: /apache
        pathType: Prefix
        backend:
          service:
            name: apache-service
            port:
              number: 80
```

Combined manifest

▼ Create a combined manifest with all the api objects

```
Copy
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:alpine
```

```
ports:
        - containerPort: 80
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
  namespace: default
spec:
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
apiVersion: apps/v1
kind: Deployment
metadata:
  name: apache-deployment
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: apache
  template:
    metadata:
      labels:
        app: apache
    spec:
      containers:
     - name: my-apache-site
        image: httpd:2.4
        ports:
        - containerPort: 80
apiVersion: v1
kind: Service
metadata:
  name: apache-service
  namespace: default
spec:
  selector:
    app: apache
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
  type: ClusterIP
apiVersion: networking.k8s.io/v1
kind: Ingress
```

```
metadata:
 name: web-apps-ingress
 namespace: default
 annotations:
   nginx.ingress.kubernetes.io/rewrite-target: /
  ingressClassName: nginx
  - host: your-domain.com
   http:
      paths:
      - path: /nginx
        pathType: Prefix
        backend:
          service:
            name: nginx-service
            port:
              number: 80
      - path: /apache
        pathType: Prefix
        backend:
          service:
            name: apache-service
            port:
              number: 80
```

Apply the manifest

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

• Update your local hosts entry (/etc/hosts) such that <u>your-domain.com</u> points to the IP of your load balancer

```
65.20.84.86 your-domain.com
```

• Try going to your-domain.com/apache and your-domain.com/nginx

Trying traefik's ingress controller

Traefik is another popular ingress controller. Let's try to our apps using it next

• Install traefik ingress controller using helm

```
Сору
  helm repo add traefik https://helm.traefik.io/traefik
  helm repo update
  helm install traefik traefi/traefik --namespace traefik --create-namespace
• Make sure an ingressClass is created for traefik
                                                       Сору
   kubectl get IngressClass
• Notice it created a LoadBalancer svc for you
                                                       Сору
   kubectl get svc -n traefik
• Create a Ingress that uses the traefik ingressClass and traefik annotations (traefik.yml)
  apiVersion: networking.k8s.io/v1
                                                       Copy
  kind: Ingress
  metadata:
    name: traefik-web-apps-ingress
    namespace: default
  spec:
    ingressClassName: traefik
    - host: traefik-domain.com
      http:
        paths:
        - path: /nginx
           pathType: Prefix
           backend:
             service:
               name: nginx-service
               port:
                 number: 80
         - path: /apache
           pathType: Prefix
           backend:
             service:
               name: apache-service
               port:
                 number: 80
• Add an entry to your /etc/hosts (IP should be your loadbalancer IP)
                                                       Сору
  65.20.90.183
                   traefik-domain.com
• Visit the website
                                                       Сору
  traefik-domain.com/nginx
```

traefik-domain.com/apache



Can you guess what is going wrong? Why are you not seeing anything on this final page?

Assignment

Try to figure out how can you rewrite the path to / if you're using traefik as the ingress class

Secrets and configmaps

Kubernetes suggests some standard configuration practises.

These include things like

- 1. You should always create a deployment rather than creating naked pods
- 2. Write your configuration files using YAML rather than JSON
- 3. Configuration files should be stored in version control before being pushed to the cluster

Kubernetes v1 API also gives you a way to store configuration of your application outside the image/pod

This is done using

- 1. ConfigMaps
- 2. Secrets

Rule of thumb

Don't bake your application secrets in your docker image

Pass them in as environment variables whenever you're starting the container

ConfigMaps

Ref - https://kubernetes.io/docs/concepts/configuration/configmap/

A ConfigMap is an API object used to store non-confidential data in key-value pairs. <u>Pods</u> can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

A ConfigMap allows you to decouple environment-specific configuration from your <u>container</u> images, so that your applications are easily portable.

Creating a ConfigMap

• Create the manifest

```
apiVersion: v1
kind: ConfigMap
metadata:
   name: ecom-backend-config
data:
   database_url: "mysql://ecom-db:3306/shop"
   cache_size: "1000"
   payment_gateway_url: "https://payment-gateway.example.com"
   max_cart_items: "50"
   session_timeout: "3600"
```

Apply the manifest

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

• Get the configmap

kubectl describe configmap ecom-backend-config

Сору

Creating an express app that exposes env variables

▼ Express app code

```
Сору
import express from 'express';
import fs from 'fs';
import path from 'path';
const app = express();
const port = 3000;
app.get('/', (req, res) => {
 const envVars = {
   DATABASE_URL: process.env.DATABASE_URL,
   CACHE_SIZE: process.env.CACHE_SIZE,
   PAYMENT_GATEWAY_URL: process.env.PAYMENT_GATEWAY_URL,
   MAX_CART_ITEMS: process.env.MAX_CART_ITEMS,
   SESSION_TIMEOUT: process.env.SESSION_TIMEOUT,
 };
 res.send(`
   <h1>Environment Variables</h1>
   ${JSON.stringify(envVars, null, 2)}
  `);
});
app.listen(port, () => {
 console.log(`App listening at http://localhost:${port}`);
});
```

▼ Dockerfile to containerise it

```
FROM node:20

WORKDIR /usr/src/app

COPY package*.json ./
RUN npm install

COPY . .

RUN npx tsc -b

EXPOSE 3000
CMD [ "node", "index.js" ]
```

▼ Deploy to dockerhub - https://hub.docker.com/repository/docker/100xdevs/env-backend/general

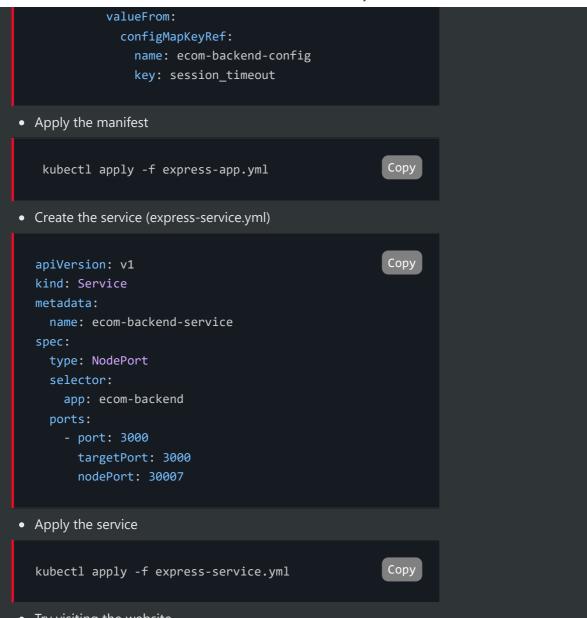
Trying the express app using docker locally

```
docker run -p 3003:3000 -e DATABASE_URL=asd 100xdevs/env-backend Copy
```

Try running using k8s locally

• Create the manifest (express-app.yml)

```
Copy
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ecom-backend-deployment
  replicas: 1
  selector:
    matchLabels:
      app: ecom-backend
  template:
    metadata:
      labels:
        app: ecom-backend
    spec:
      containers:
      - name: ecom-backend
        image: 100xdevs/env-backend
        ports:
        - containerPort: 3000
        env:
        - name: DATABASE URL
          valueFrom:
            configMapKeyRef:
              name: ecom-backend-config
              key: database url
        - name: CACHE_SIZE
          valueFrom:
            configMapKeyRef:
              name: ecom-backend-config
              key: cache_size
        - name: PAYMENT GATEWAY URL
          valueFrom:
            configMapKeyRef:
              name: ecom-backend-config
              key: payment_gateway_url
        - name: MAX_CART_ITEMS
          valueFrom:
            configMapKeyRef:
              name: ecom-backend-config
              key: max_cart_items
        - name: SESSION_TIMEOUT
```



• Try visiting the website

Secrets

Secrets are also part of the kubernetes v1 api. They let you store passwords / sensitive data which can then be mounted on to pods as environment variables. Using a Secret means that you don't need to include confidential data in your application code.

Ref - https://kubernetes.io/docs/concepts/configuration/secret/

Using a secret

 Create the manifest with a secret and pod (secret value is base64 encoded) (https://www.base64encode.org/)

```
Coi
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
  .env: REFUQUJBU0VfVVJMPSJwb3N0Z3JlczovL3VzZXJuYW110nNlY3JldEBsb2NhbGhvc3QvcG9zdGdy
apiVersion: v1
kind: Pod
metadata:
  name: secret-dotfiles-pod
spec:
  containers:
    - name: dotfile-test-container
      image: nginx
      volumeMounts:
        - name: env-file
          readOnly: true
          mountPath: "/etc/secret-volume"
  volumes:
    - name: env-file
      secret:
        secretName: dotfile-secret
```

• Try going to the container and exploring the .env

```
kubectl exec -it secret-dotfiles-pod /bin/bash
cd /etc/secret-volume/
ls
```

Base64 encoding

Whenever you're storing values in a secret, you need to base64 encode them. They can still be decoded, and hence this is not for security purposes. This is more to provide a standard way to store secrets, incase they are binary in nature.

For example, TLS (https) certificates that we'll be storing as secrets eventually can have non ascii characters. Converting them to base64 converts them to ascii characters.

Secrets as env variables

You can also pass in secrets as environment variables to your process (similar to how we did it for configmaps in the last slide)

• Create the secret

```
Сору
  apiVersion: v1
  kind: Secret
  metadata:
    name: my-secret
    username: YWRtaW4= # base64 encoded 'admin'
    password: cGFzc3dvcmQ= # base64 encoded 'password'
• Create the pod
                                                                                           Cor
  apiVersion: v1
  kind: Pod
  metadata:
    name: secret-env-pod
  spec:
    containers:
    - name: my-container
      image: busybox
      command: ["/bin/sh", "-c", "echo Username: $USERNAME; echo Password: $PASSWORD;
      - name: USERNAME
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: username
      - name: PASSWORD
        valueFrom:
          secretKeyRef:
            name: my-secret
            key: password
```

ConfigMaps vs Secrets

• Creating a ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
   name: example-config
data:
   key1: value1
   key2: value2
```

Creating a Secret

apiVersion: v1
kind: Secret
metadata:
 name: example-secret
data:
 password: cGFzc3dvcmQ=
 apiKey: YXBpa2V5

Key differences

• Purpose and Usage:

- **Secrets**: Designed specifically to store sensitive data such as passwords, OAuth tokens, and SSH keys.
- **ConfigMaps:** Used to store non-sensitive configuration data, such as configuration files, environment variables, or command-line arguments.

• Base64 Encoding:

- Secrets: The data stored in Secrets is base64 encoded. This is not encryption but simply encoding, making it slightly obfuscated. This encoding allows the data to be safely transmitted as part of JSON or YAML files.
- ConfigMaps: Data in ConfigMaps is stored as plain text without any encoding.

• Volatility and Updates:

- **Secrets**: Often, the data in Secrets needs to be rotated or updated more frequently due to its sensitive nature.
- ConfigMaps: Configuration data typically changes less frequently compared to sensitive data.

Kubernetes Features:

- Secrets: Kubernetes provides integration with external secret management systems and supports encryption at rest for Secrets when configured properly. Ref https://secrets-store-csi-driver.sigs.k8s.io/concepts.html#provider-for-the-secrets-store-csi-driver
- ConfigMaps: While ConfigMaps are used to inject configuration data into pods, they do not have the same level of support for external management and encryption.

Adding https using cert-manager

Ref - https://cert-manager.io/

Try installing a certificate for a domain name of your own before tomorrows class

Maybe get a domain name from namecheap for cheap - https://www.namecheap.com/

Volumes in docker

Pretext

The following docker image runs a Node.js app that writes peridically to the filesystem -

https://hub.docker.com/r/100xdevs/write-random

▼ Nodejs Code

```
Copy
const fs = require('fs');
const path = require('path');
// Function to generate random data
function generateRandomData(length) {
    let characters = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrst
    let result = '';
    for (let i = 0; i < length; i++) {</pre>
        result += characters.charAt(Math.floor(Math.random() * chara
    return result;
// Write random data to a file
function writeRandomDataToFile(filePath, dataLength) {
    const data = generateRandomData(dataLength);
    fs.writeFile(filePath, data, (err) => {
        if (err) {
            console.error('Error writing to file', err);
        } else {
            console.log('Data written to file', filePath);
    });
// Define the file path and data length
const filePath = path.join(__dirname, '/generated/randomData.txt');
const dataLength = 100; // Change this to desired length of random d
// Write random data to file every 10 seconds
setInterval(() => {
   writeRandomDataToFile(filePath, dataLength);
}, 10000); // 10000 ms = 10 seconds
// Keep the script running
console.log('Node.js app is running and writing random data to rando
```

Run it in docker

Try running the image above in your local machine



Try going to the container and seeing the contents of the container

docker exec -it container_id /bin/bash
cat randomData.txt
Copy

Where is this file being stored?

The data is stored in the docker runtime filesystem . When the container dies, the data dies with it. This is called ephemeral storage

Volumes in docker

If you want to persist data across container stops and starts, you can use Volumes in Docker

Bind mounts

Replace the mount on the left with a folder on your own machine

docker run -v /Users/harkiratsingh/Projects/100x/mount:/usr/src/app/generated 100xde

Volume Mounts

Create a volume

docker volume create hello Copy

• Mount data to volume

docker run -v hello:/usr/src/app/generated 100xdevs/write-random Copy

If you stop the container in either case, the randomFile.txt file persists

Volumes in kubernetes

Ref - https://kubernetes.io/docs/concepts/storage/volumes/

Volumes

In Kubernetes, a Volume is a directory, possibly with some data in it, which is accessible to a Container as part of its filesystem. Kubernetes supports a variety of volume types, such as EmptyDir, PersistentVolumeClaim, Secret, ConfigMap, and others.

Why do you need volumes?

- If two containers in the same pod want to share data/fs.
- If you want to create a database that persists data even when a container restarts (creating a DB)
- Your pod just needs extra space during execution (for caching lets say) but doesnt care if it persists or not.

Types of volumes

Ephemeral Volume

Temporary volume that can be shared amongst various containers of a pod. When the pods dies, the volume dies with it.

For example -

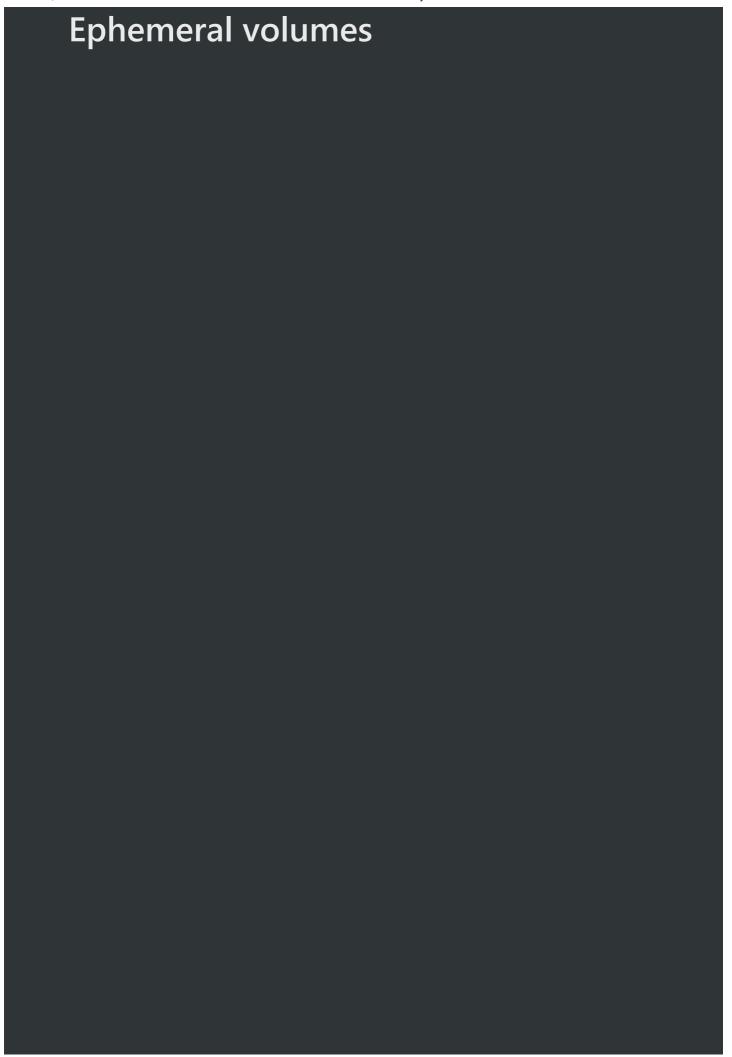
- 1. ConfigMap
- 2. Secret
- 3. emptyDir

Persistent Volume

A Persistent Volume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

Persistent volume claim

A Persistent Volume Claim (PVC) is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).



A lot of times you want two containers in a pod to share data. But when the pods dies, then the data can die with it.

Persistent volumes

Just like our kubernetes cluster has nodes where we provision our pods .

We can create peristent volumes where our pods can claim (ask for) storage

Persistent volumes can be provisioned statically or dynamically.

Static persistent volumes

Creating a NFS

NFS is one famous implementation you can use to deploy your own persistent volume

I'm running one on my aws server -

```
version: '3.7'

services:
    nfs-server:
    image: itsthenetwork/nfs-server-alpine:latest
    container_name: nfs-server
    privileged: true
    environment:
        SHARED_DIRECTORY: /exports
    volumes:
        - ./data:/exports:rw
    ports:
        - "2049:2049"
    restart: unless-stopped

Make sure the 2049 port on your machine is open
```

Creating a pv and pvc

Create a persistent volume claim and persistent volume

```
apiVersion: v1
kind: PersistentVolume
metadata:
   name: nfs-pv
spec:
   capacity:
    storage: 10Gi
   accessModes:
        - ReadWriteMany
   storageClassName: nfs
   nfs:
        path: /exports
        server: 52.66.197.168
---
apiVersion: v1
Copy

C
```

```
kind: PersistentVolumeClaim
metadata:
    name: nfs-pvc
spec:
    accessModes:
    - ReadWriteMany
    resources:
        requests:
        storage: 10Gi
    storageClassName: nfs
```

Create a pod

```
Сору
apiVersion: v1
kind: Pod
metadata:
  name: mongo-pod
spec:
  containers:
  - name: mongo
    image: mongo:4.4
    command: ["mongod", "--bind_ip_all"]
    ports:
    - containerPort: 27017
    volumeMounts:
    - mountPath: "/data/db"
      name: nfs-volume
  volumes:
  - name: nfs-volume
    persistentVolumeClaim:
      claimName: nfs-pvc
```

Try it out

• Put some data in mongodb

```
kubectl exec -it mongo-pod -- mongo
use mydb
db.mycollection.insert({ name: "Test", value: "This is a test" })
exit
```

• Delete and restart the pod

```
kubectl delete pod mongo-pod
kubectl apply -f mongo.yml
```

• Check if the data persists

```
kubectl exec -it mongo-pod -- mongo
use mydb
db.mycollection.find()
```

Automatic pv creation

Ref - https://docs.vultr.com/how-to-provision-persistent-volume-claims-on-vultr-kubernetes-engine

• Create a persistent volume claim with storageClassName set to vultr-block-storage-hdd

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
   name: csi-pvc
spec:
   accessModes:
    - ReadWriteOnce
   resources:
     requests:
        storage: 40Gi
   storageClassName: vultr-block-storage-hdd
```

• Apply the pod manifest

```
Сору
apiVersion: v1
kind: Pod
metadata:
  name: mongo-pod
spec:
  containers:
  - name: mongo
    image: mongo:4.4
    command: ["mongod", "--bind_ip_all"]
    - containerPort: 27017
    volumeMounts:
    - name: mongo-storage
      mountPath: /data/db
  volumes:
  - name: mongo-storage
```

