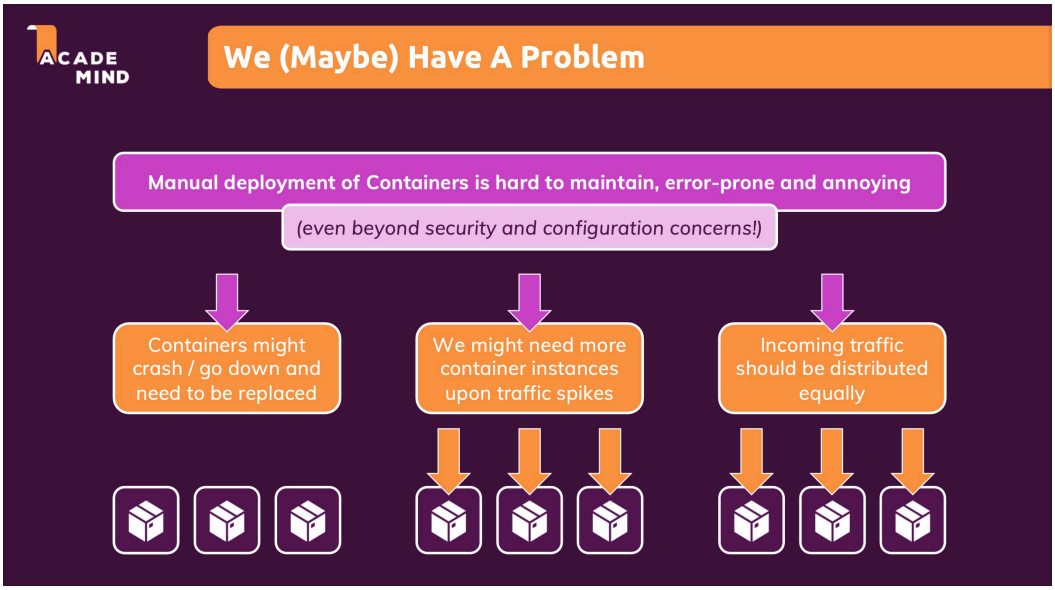
Kubernetes Course:

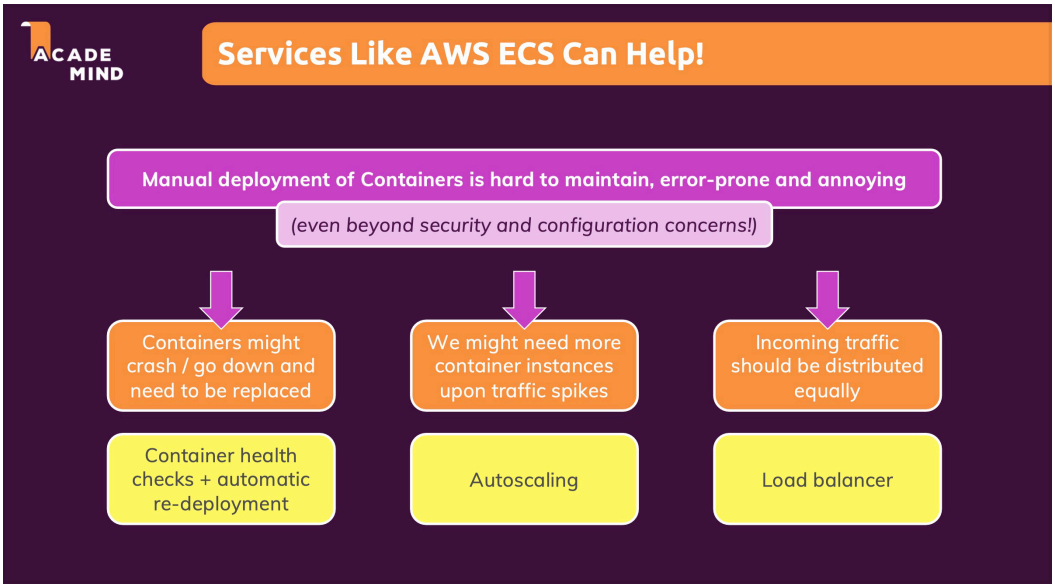
Section 11: Getting Started with Kubernetes:

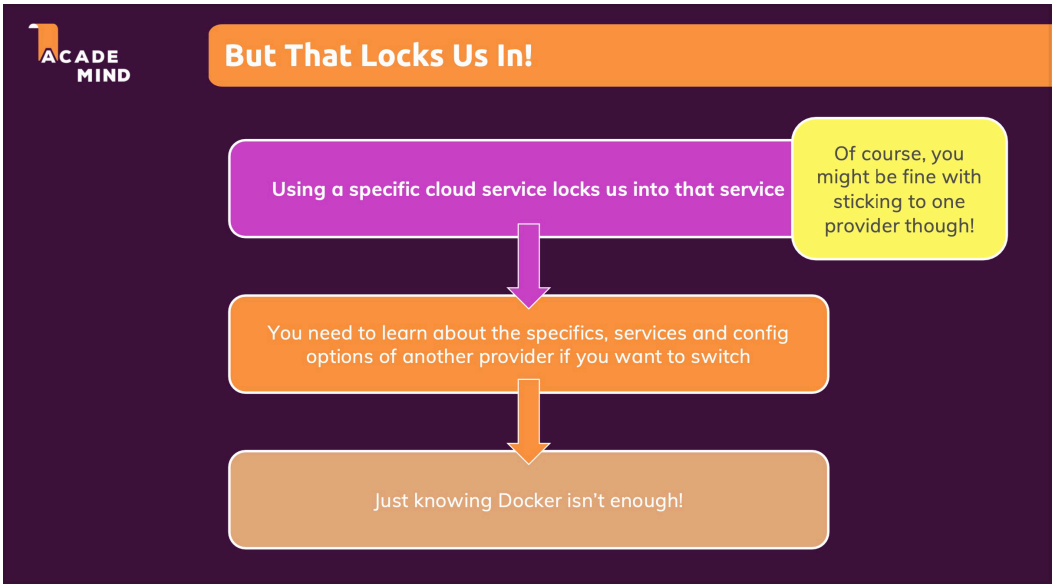
More Problems with Manual Deployment:

Kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

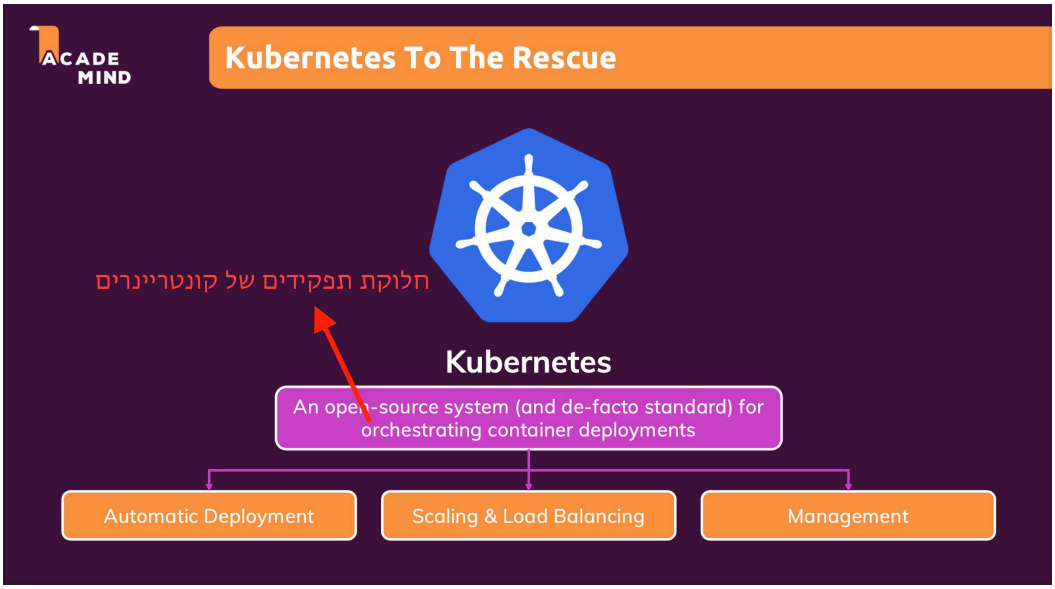


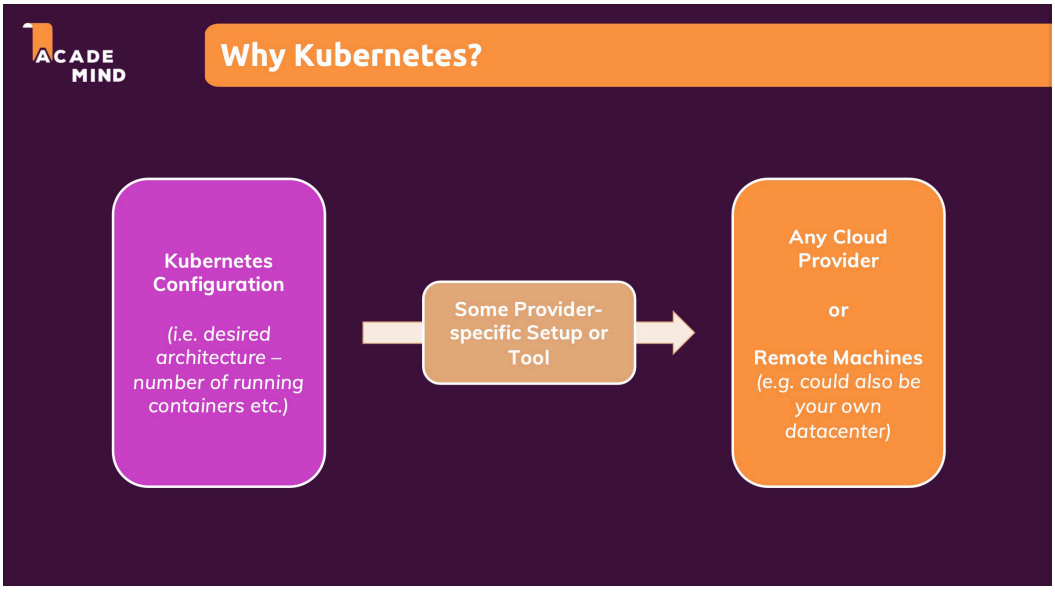
Why Kubernetes?:





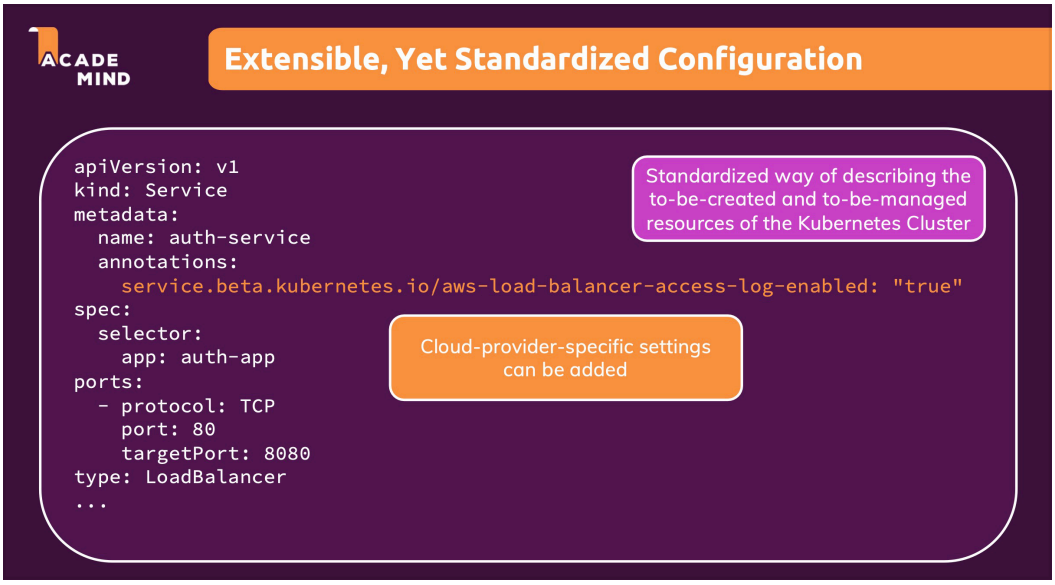
What Is Kubernetes Exactly?:



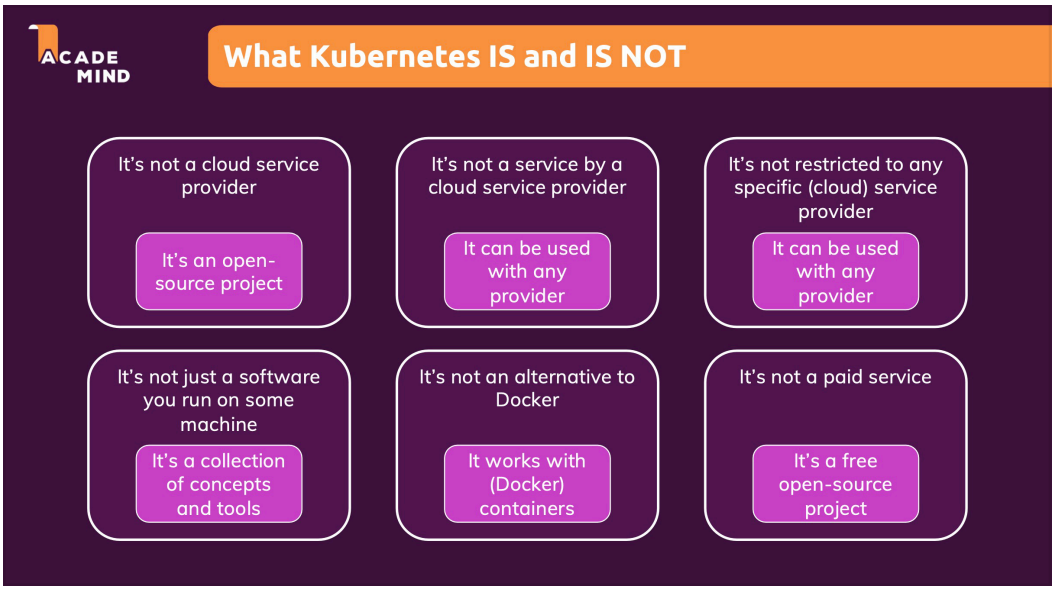


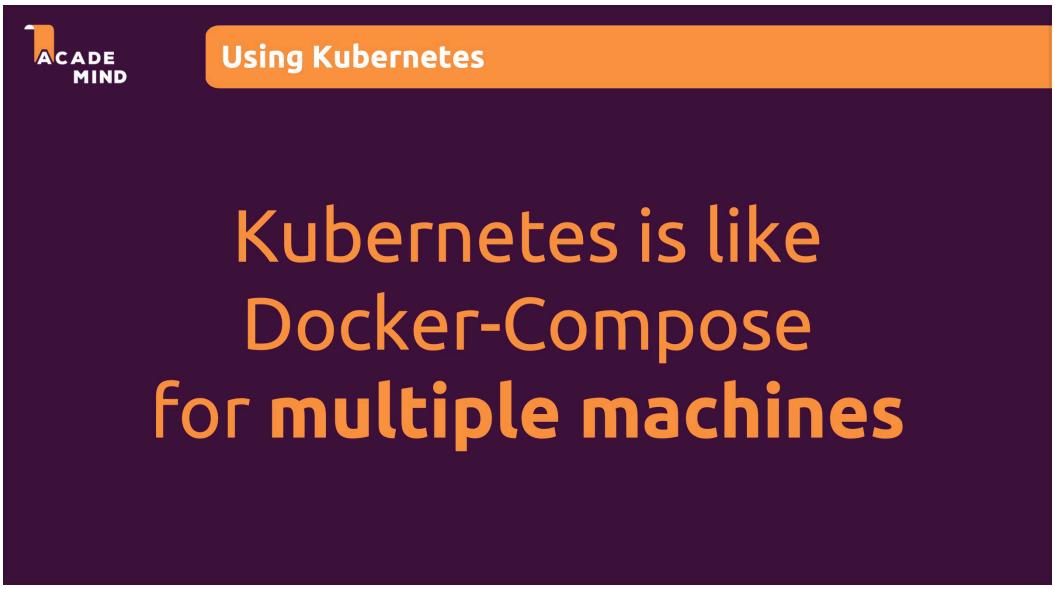
Kubernetes allows us to write down a configuration file and, then pass the

configuration file with certain tools to any cloud provider or any machine owned by us.
Which can then pick up the setup we want.
We have one way to setup a configuration file and, this configuration will work on any cloud provider.

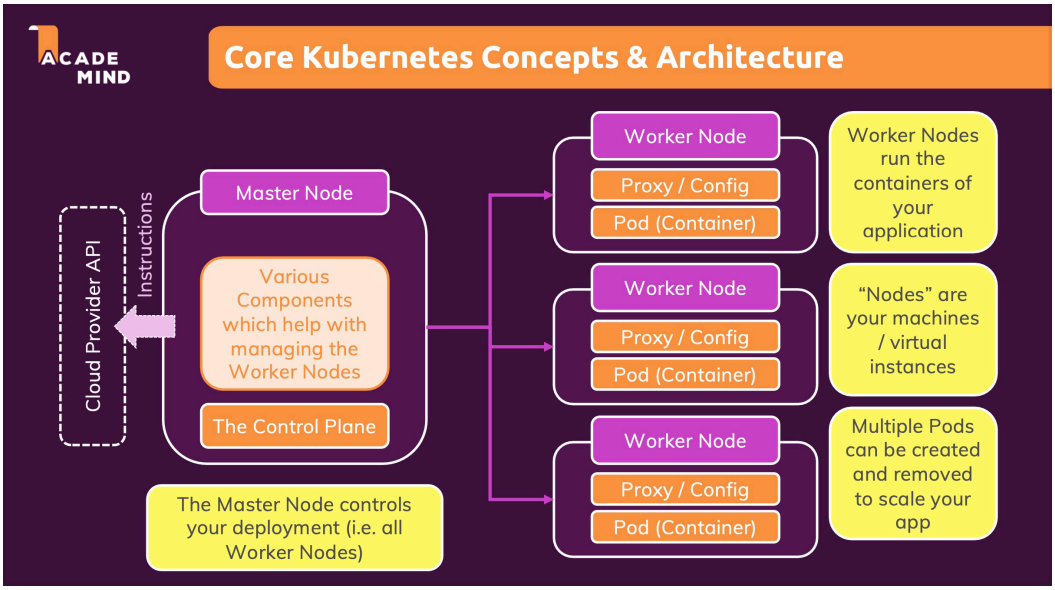


The idea behind Kubernetes - **Having a standardize way of deployment.**

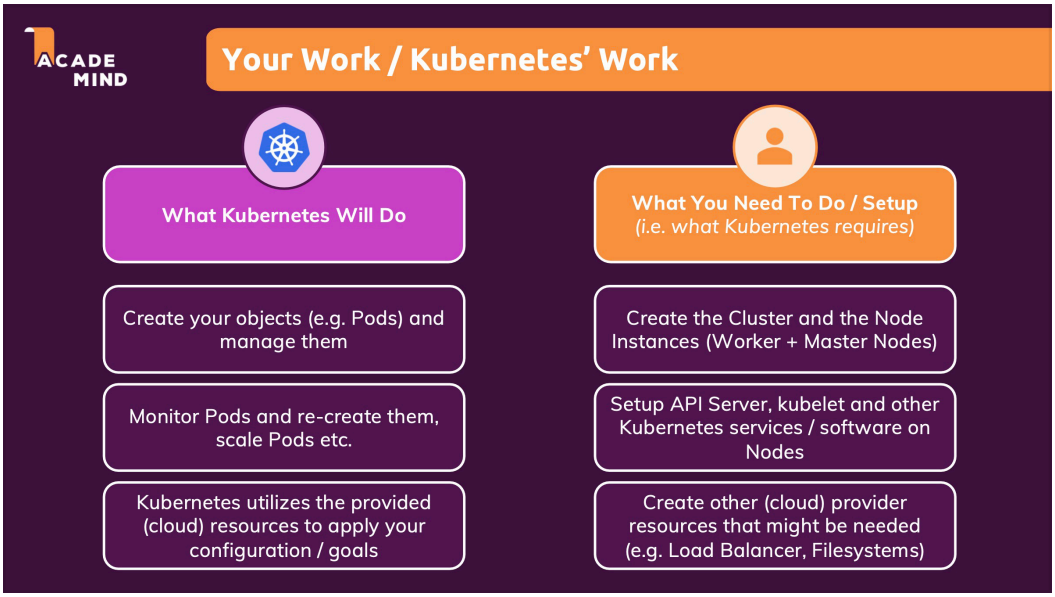




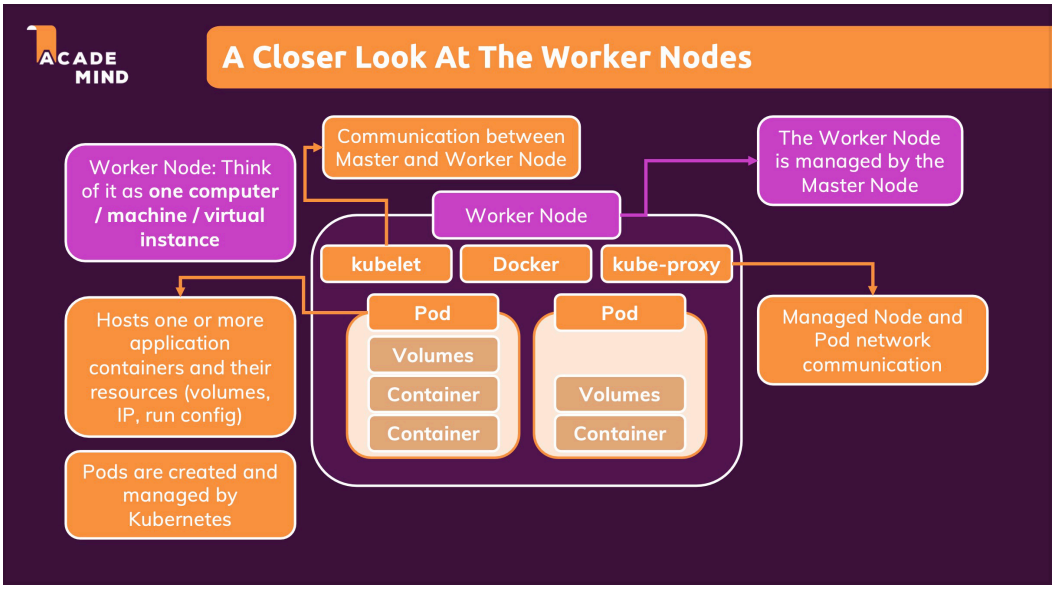
Kubernetes: Architecture & Core Concepts:



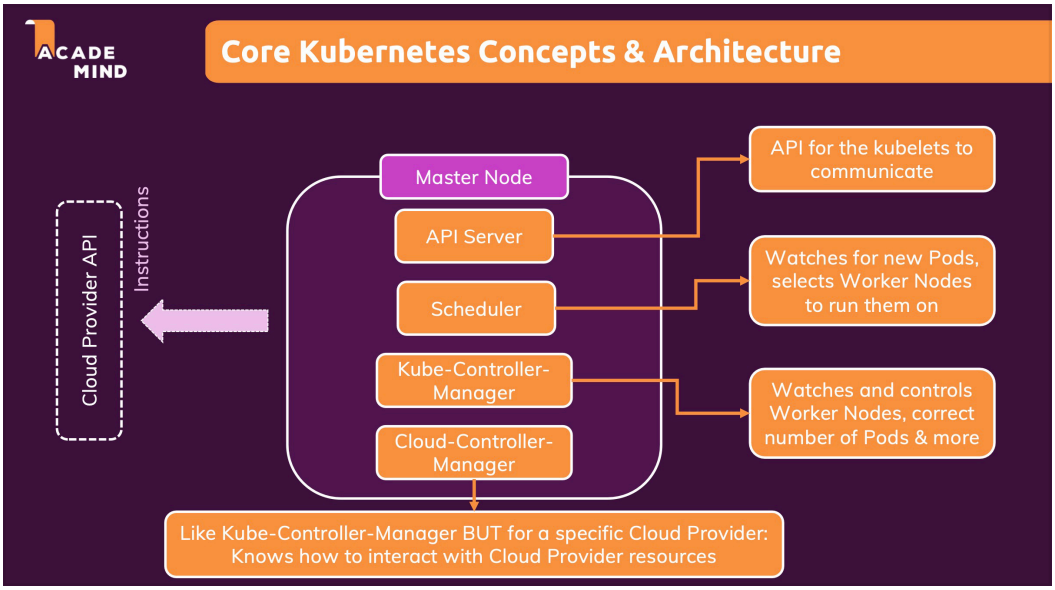
Kubernetes will NOT manage your Infrastructure!:



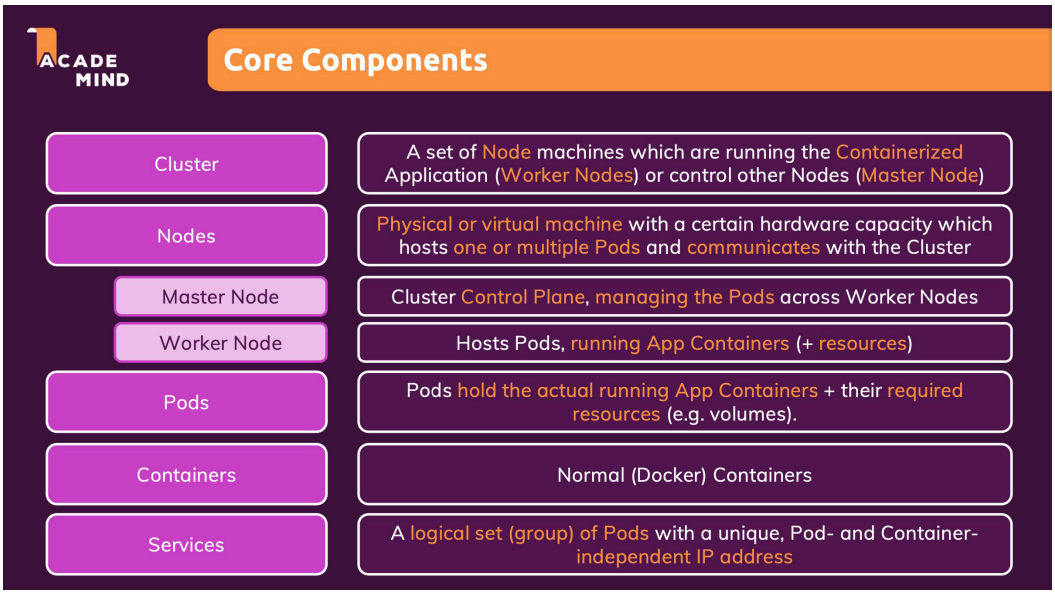
A Closer Look at the Worker Nodes:



A Closer Look at the Master Node:



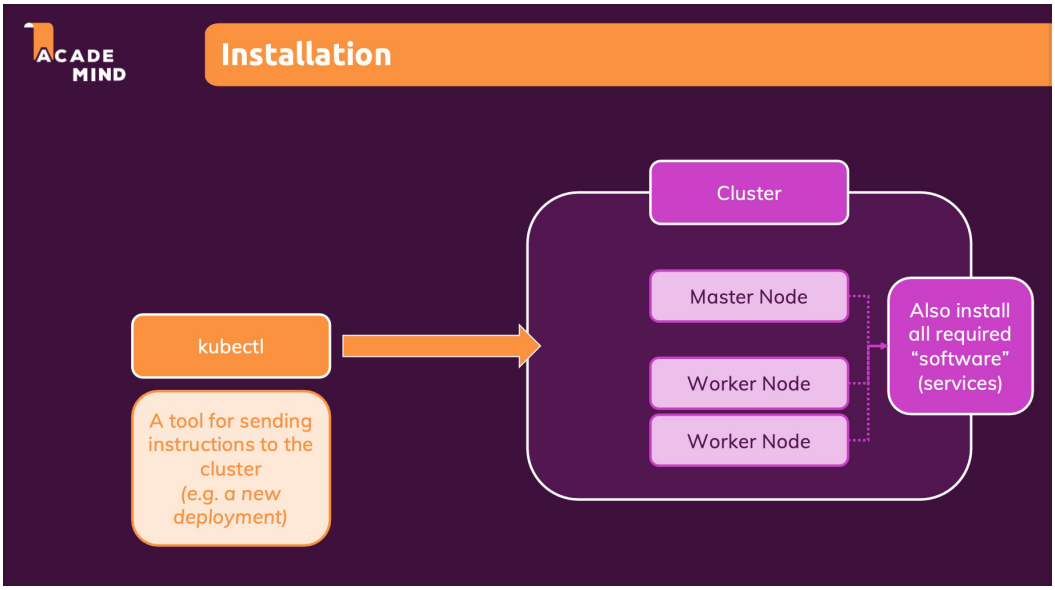
Important Terms & Concepts:



Section 12: Kubernetes in Action - Diving into the Core Concepts:

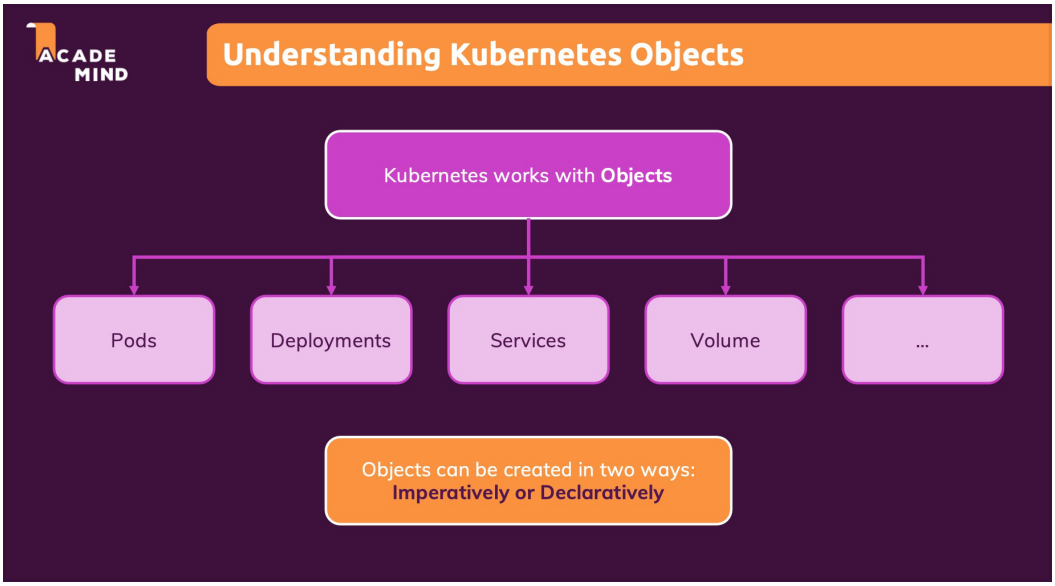
Kubernetes: Required Setup & Installation Steps:

In order to work with Kubernetes we first we need to install
It on the machine we want to use it.
"Kubectl (kube control)" - is a tool to send instructions to the cluster.
We as a developers use the Kubecntl to send instructions to the master node - telling the master node what to do with worker nodes.
Minikube installation setup - https://minikube.sigs.k8s.io/docs/start/ .
Minikube is a tool we install on our local machine in order to setup a dummy cluster that we can work on and test (its another machine - virtual machine).
Kubectl installation setup - https://kubernetes.io/docs/tasks/tools/ .

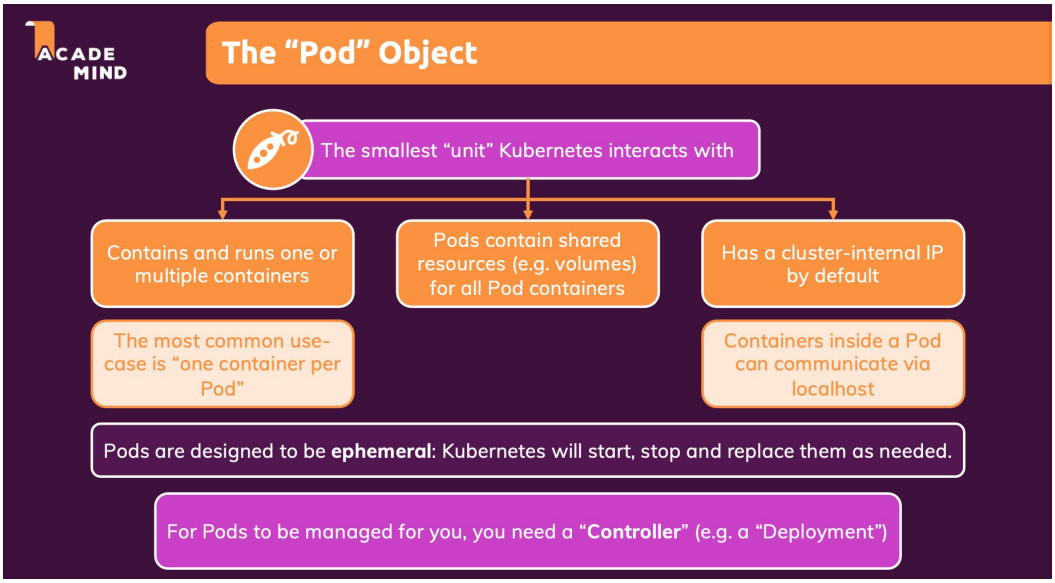


macOS Setup:
https://www.udemy.com/course/docker-kubernetes-the-practical-guide/learn/lecture/22627609#overview

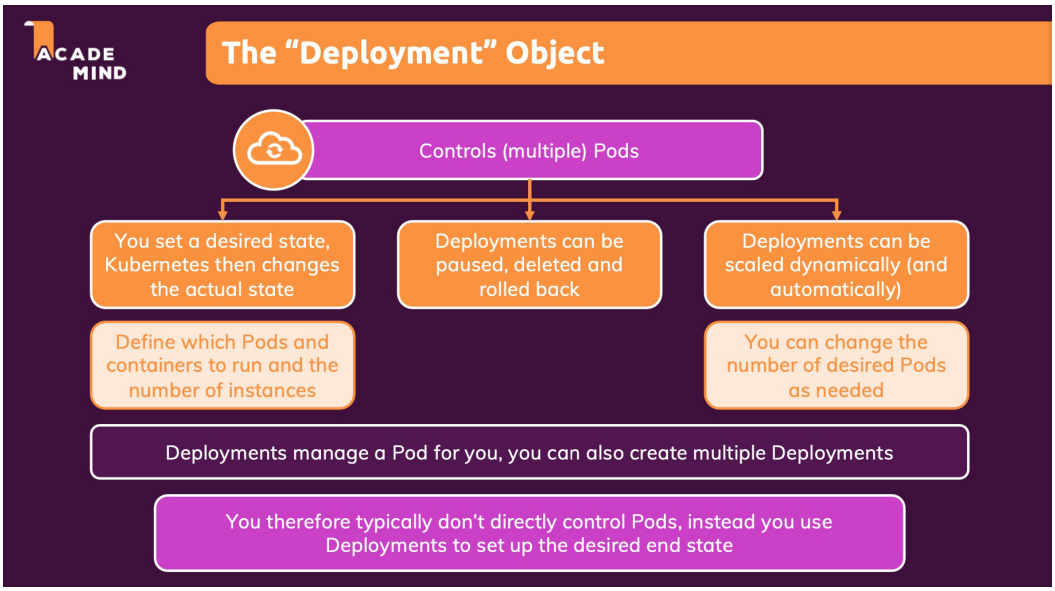
Understanding Kubernetes Objects (Resources):



The "Pod" object:



The "Deployment" Object (Resource):
The main object we will work with.
The deployment object is able to control one or more pods.

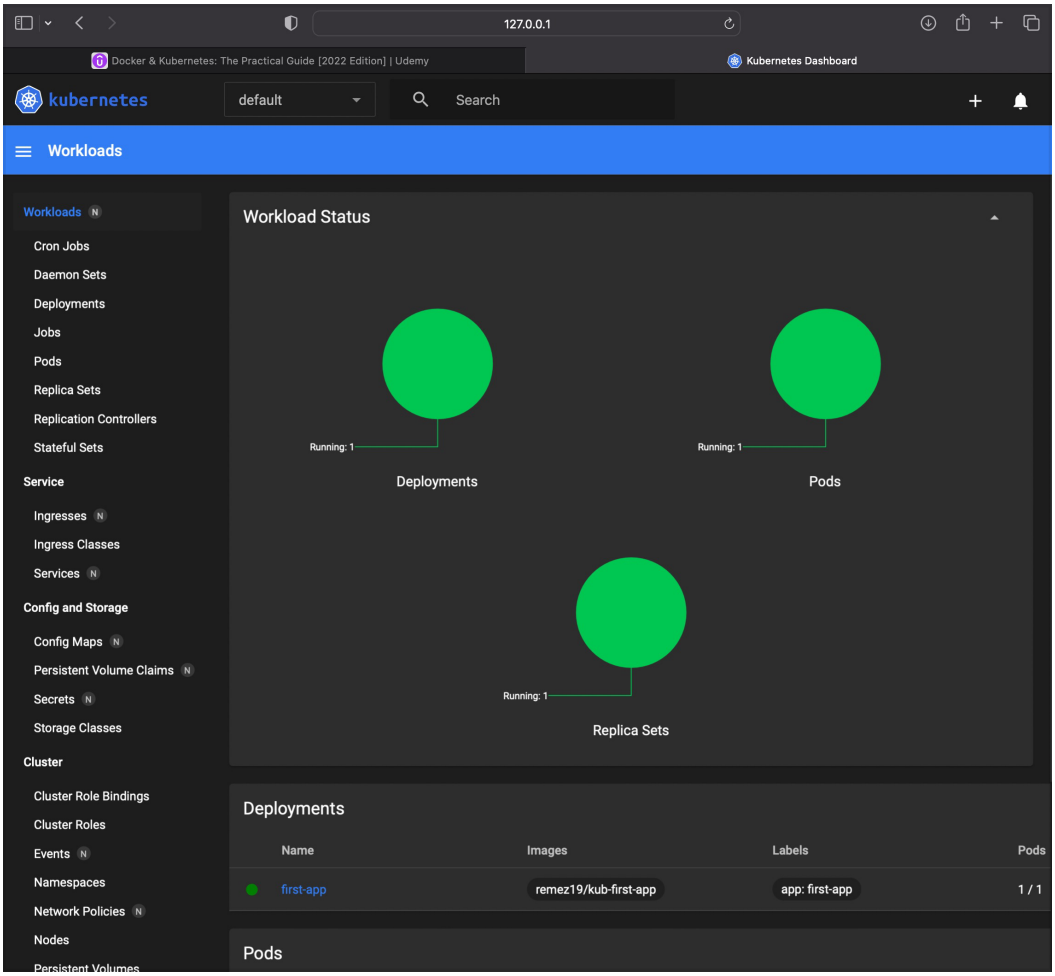


A First Deployment - Using the Imperative Approach:

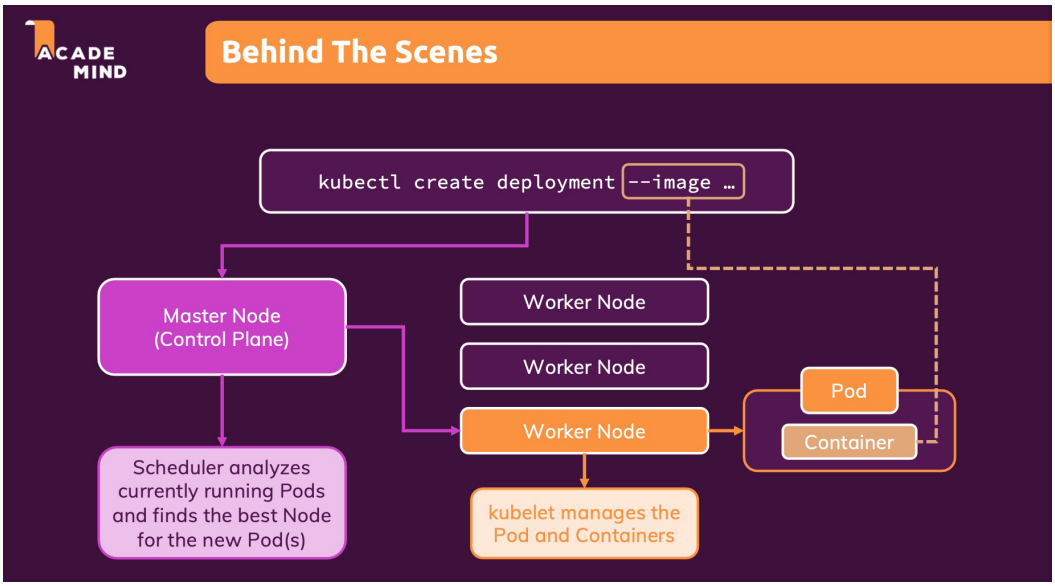
- The first step is to create the image of our application (we want to deploy).
- The second step is to send this image to the cluster by creating a deployment object.
  To achieve this we use the command - "kubectl create deployment [OBJECT_NAME] —image=[IMAGE_NAME]".
  With the "—image" option we specify which image should be used for the container of the pod created in this deployment.
  The image should be available in docker hub (the cluster will look for the image in docker hub).
  By default this object is sent to the Kubernetes cluster.

Example: “**kubectl create deployment first-app --image=remez19/kub-first-app**”

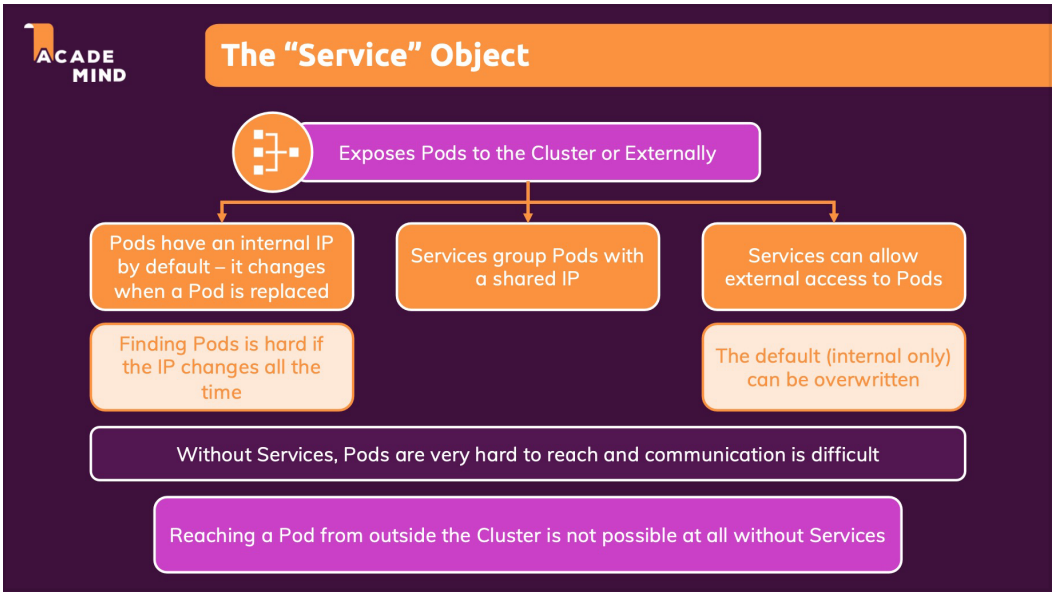
- With “minikube dashboard” we can see the cluster.



kubectl: Behind The Scenes:

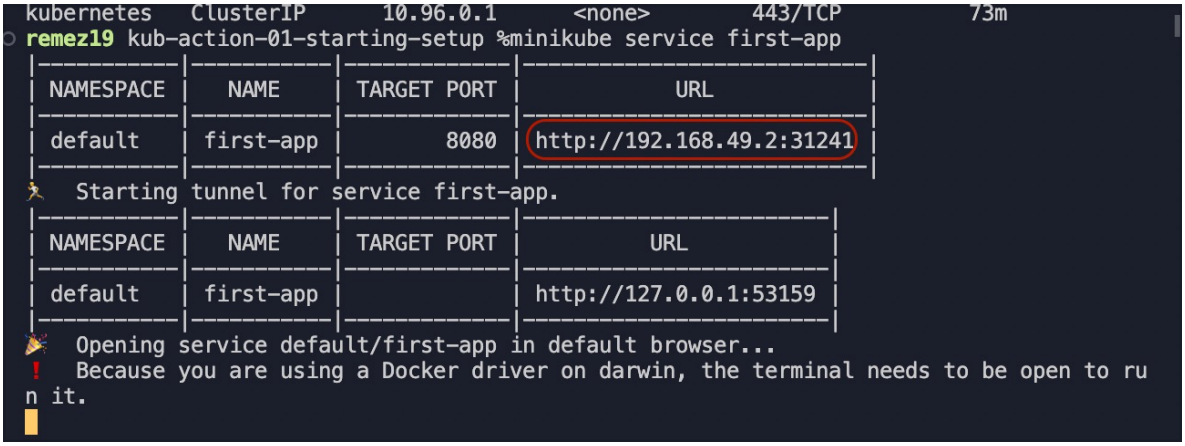


The "Service" Object (Resource):



Exposing a Deployment with a Service:

- The “kubectl expose” command exposes a pod created by deployment by **creating a service.**
- **Example**: “**kubectl expose deployment first-app --type=LoadBalancer --port=8080**”.
- We can see that this service is created by “kubectl get services”.
- To get the ip of the pod (see our running container): “kubectl service [SERVICE_NAME]”.
- **Example**:



Restarting Containers:
Because we used the deployment object we get a behavior - if the container crashed and therefor the pod crash (in which this container is running) then the pod will automatically restart therefor the container will restart.

Scaling in Action:
"**kubectl scale deployment/first-app --replicas=3**" - will create 3 pods (of the same image).
"**kubectl scale deployment/first-app --replicas=1**" - will scale down (1 pod).

Updating Deployments:

- In order to update a deployment we first need to **rebuild our image and give it a different tag**.
- Second step is to **push the image to docker hub**.
- Then we can use: "kubectl set image deployment/ [DEPLOYMENT_NAME] [CONTAINER_NAME]=[IMAGE_NAME_DOCKER_HUB]"
- **Example**: "**kubectl set image deployment/first-app kub-first-app=remez19/kub-first-app:2**".
- We can see the actions in the "Events" tab inside the Kubernetes dashboard.

Events

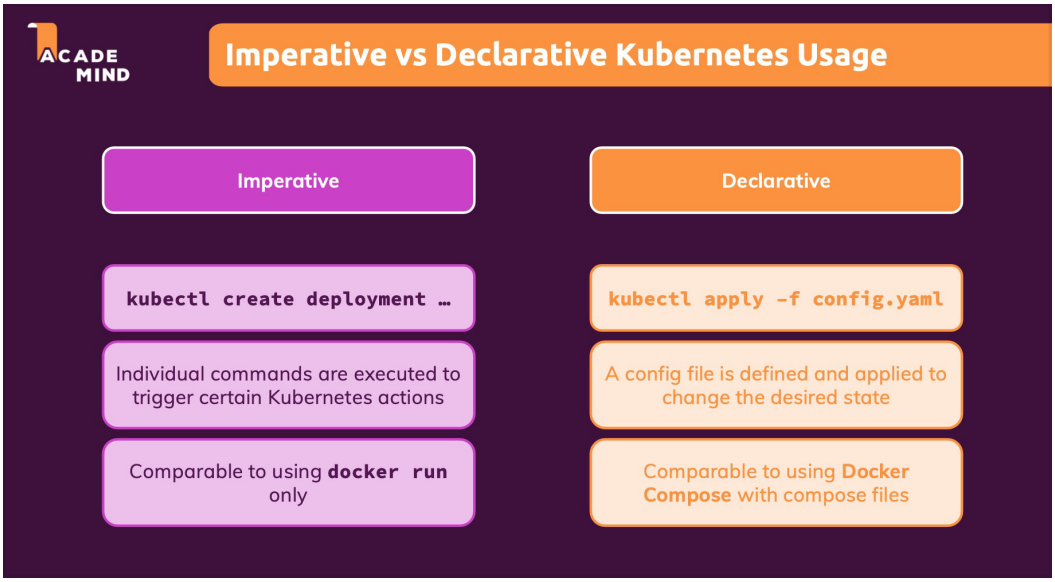
Name	Reason	Message	Source	Sub-object	Count
first-app-5f99f8d677-mplv9.171d492850bf8e	Pulled	Successfully pulled image "remez19/kub-first-app:2" in 4.023206086s	kubelet minikube	spec.containers{kub-first-app}	1
first-app-5f99f8d677-mplv9.171d49285266e5	Created	Created container kub-first-app	kubelet minikube	spec.containers{kub-first-app}	1
first-app-5f99f8d677-mplv9.171d4928568f09	Started	Started container kub-first-app	kubelet minikube	spec.containers{kub-first-app}	1
first-app-5f99f8d677-mplv9.171d492743f7f9	Scheduled	Successfully assigned default/first-app-5f99f8d677-mplv9 to minikube	default-scheduler	-	1
first-app-5f99f8d677-mplv9.171d492760f220	Pulling	Pulling image "remez19/kub-first-app:2"	kubelet minikube	spec.containers{kub-first-app}	1

Deployment Rollbacks & History:

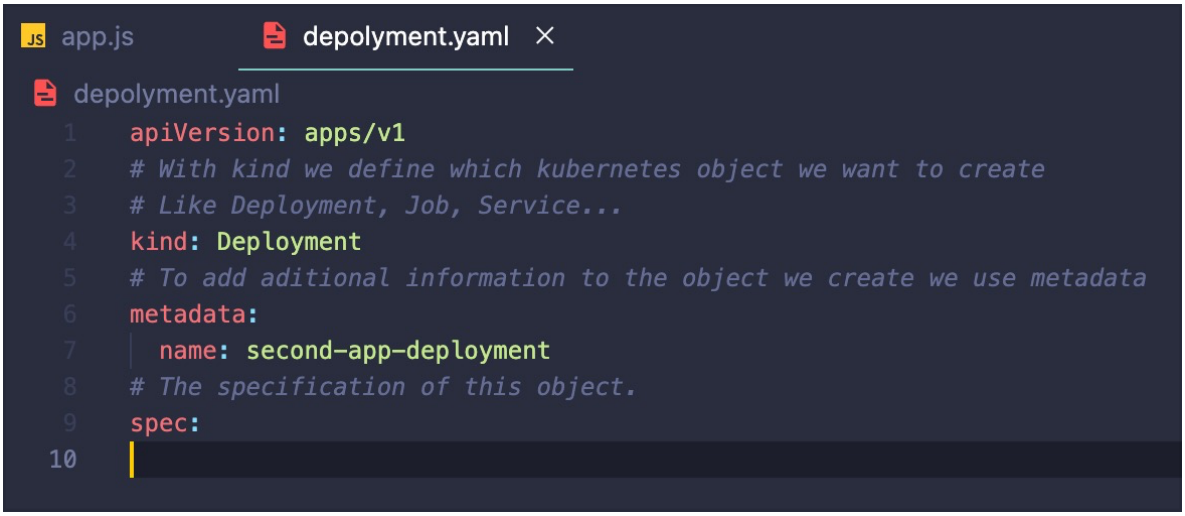
In cases where we don't want Kubernetes to continue with an update we can use rollback.
"**Kubectl rollout undo deployment/[DEPLOYMENT_NAME]**" - will undo the latest deployment.
If want to go to an older deployment (not the latest one), we can first see our deployment history: "**kubectl rollout history deployment/[DEPLOYMENT_NAME]**".
Than we can get detaild about a specific deployment by using the "REVISION" like: "**kubectl rollout history deployment/[DEPLOYMENT_NAME] —REVISION=[REVISION_NUM]**"
Now to rollout to a specific deployment we can use the revision as such: "**Kubectl rollout undo deployment/[DEPLOYMENT_NAME] —to-revision=[REVISION_NUM]**".

The Imperative vs The Declarative Approach:

Kubernetes allows us to create resource definition files.



Creating a Deployment Configuration File (Declarative Approach):



Adding Pod and Container Specs:

app.js depolyment.yaml

depolyment.yaml

```yaml
apiVersion: apps/v1
# With kind we define which kubernetes object we want to create
# Like Deployment, Job, Service...
kind: Deployment
# metadata - To add aditional information to the object
metadata:
  # name - the name of the object.
  name: second-app-deployment
# spec - specification of this object.
spec:
  # replicas - the number of pods we want (by default 1).
  replicas: 1
  # template - with this we define the pods that should be created as
  # part of this deployment.
  # template is always a pod.
  template:
    metadata:
      lables:
        # its up to us to choose the name of the key & value
        app: second-app
    spec:
      # container allow us to define the containers or container
      # that should be part of this pod.
      container:
        - name: second-node
          image: remez19/kub-first-app:2

```

Continue:

d.yaml

```yaml
apiVersion: apps/v1
# With kind we define which kubernetes object we want to create
# Like Deployment, Job, Service...
kind: Deployment
# metadata - To add aditional information to the object
metadata:
  # name - the name of the object.
  name: second-app-deployment
# spec - specification of this object.
spec:
  # replicas - the number of pods we want (by default 1).
  replicas: 1
  # With the selector we specify wich pods sholud be managed by this deployment.
  selector:
    matchLabels:
      app: second-app
  # template - with this we define the pods that should be created as
  # part of this deployment.
  # template is always a pod.
  template:
    metadata:
      lables:
        # its up to us to choose the name of the key & value
        app: second-app
    spec:
      # container allow us to define the containers or container
      # that should be part of this pod.
      container:
        - name: second-node
          image: remez19/kub-first-app:2
```

Tells the deployment which pods belong to it.

To use the file - "kubectl apply -f=[PATH_TO_FILE]".
Example - "kubectl apply -f=d.yaml"

Creating a Service Declaratively:

service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: backend
spec:
  selector:
    # Which pods should be managed by this service
    app: second-app
  ports:
    - protocol: "TCP"
      port: 80
      targetPort: 8080
  type: LoadBalancer
```

Using this file - "**kubectl apply -f=service.yaml**"

Updating & Deleting Resources:

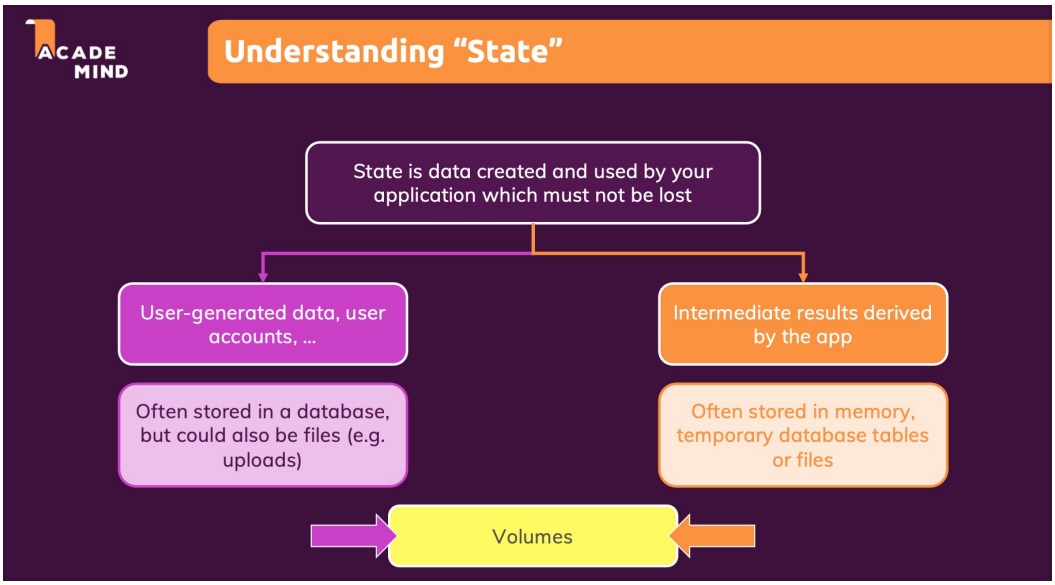
To make changes we can change our file however we want (create any change we want) and, run "**kubectl apply -f=service.yaml**".

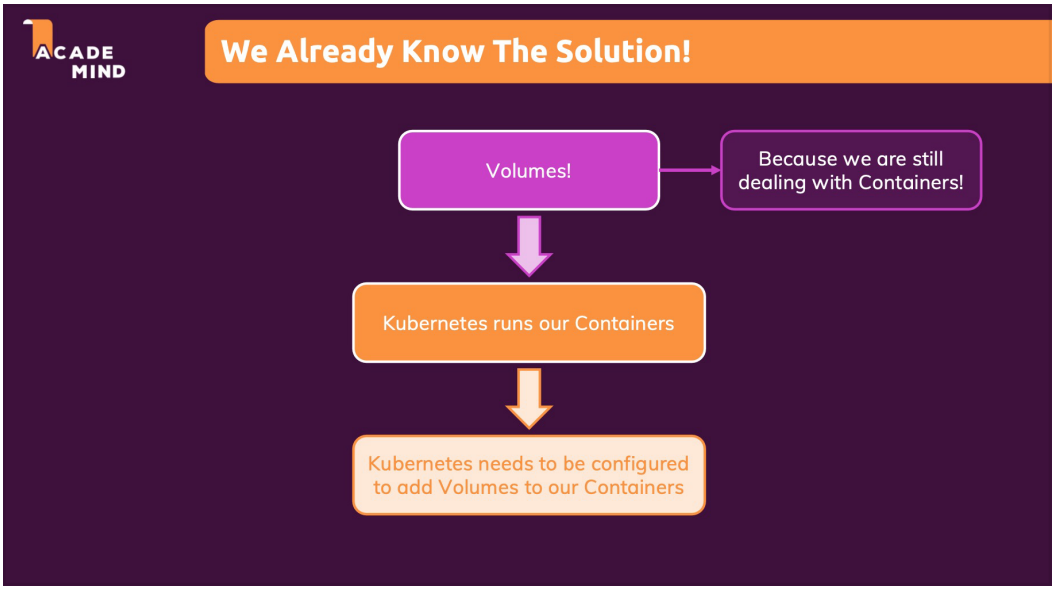
Multiple vs Single Config Files:

Merging the two file into one file.
Separating the objects with "—"

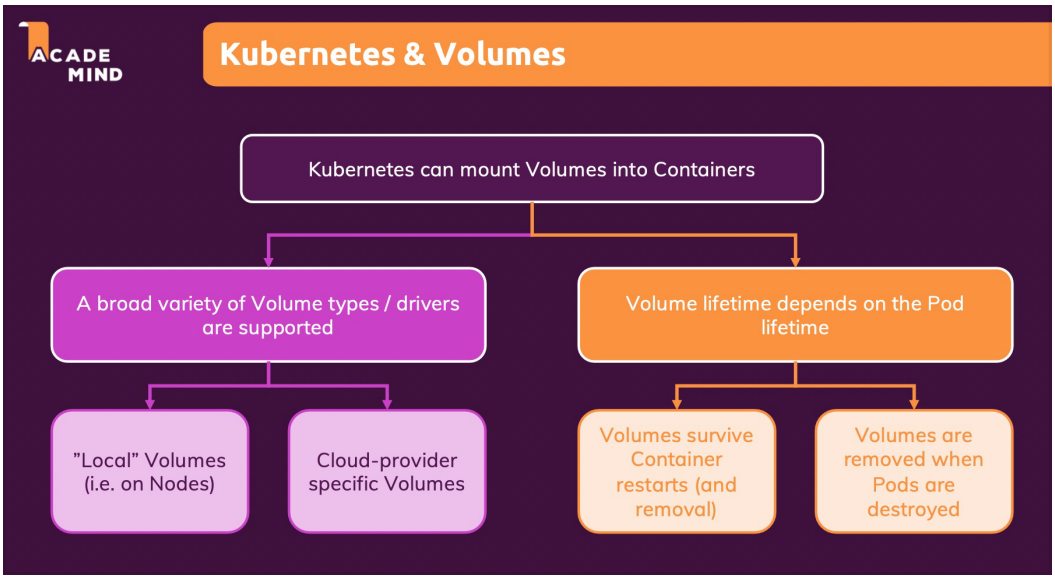
Section 13: Managing Data & Volumes with Kubernetes:

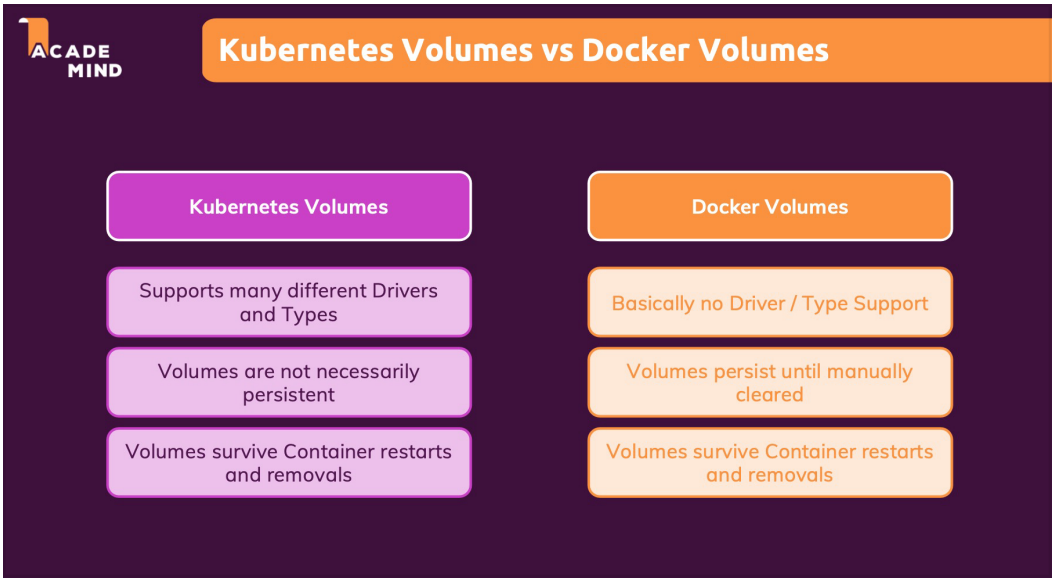
Kubernetes & Volumes - More Than Docker Volumes:





Kubernetes Volumes: Theory & Docker Comparison:



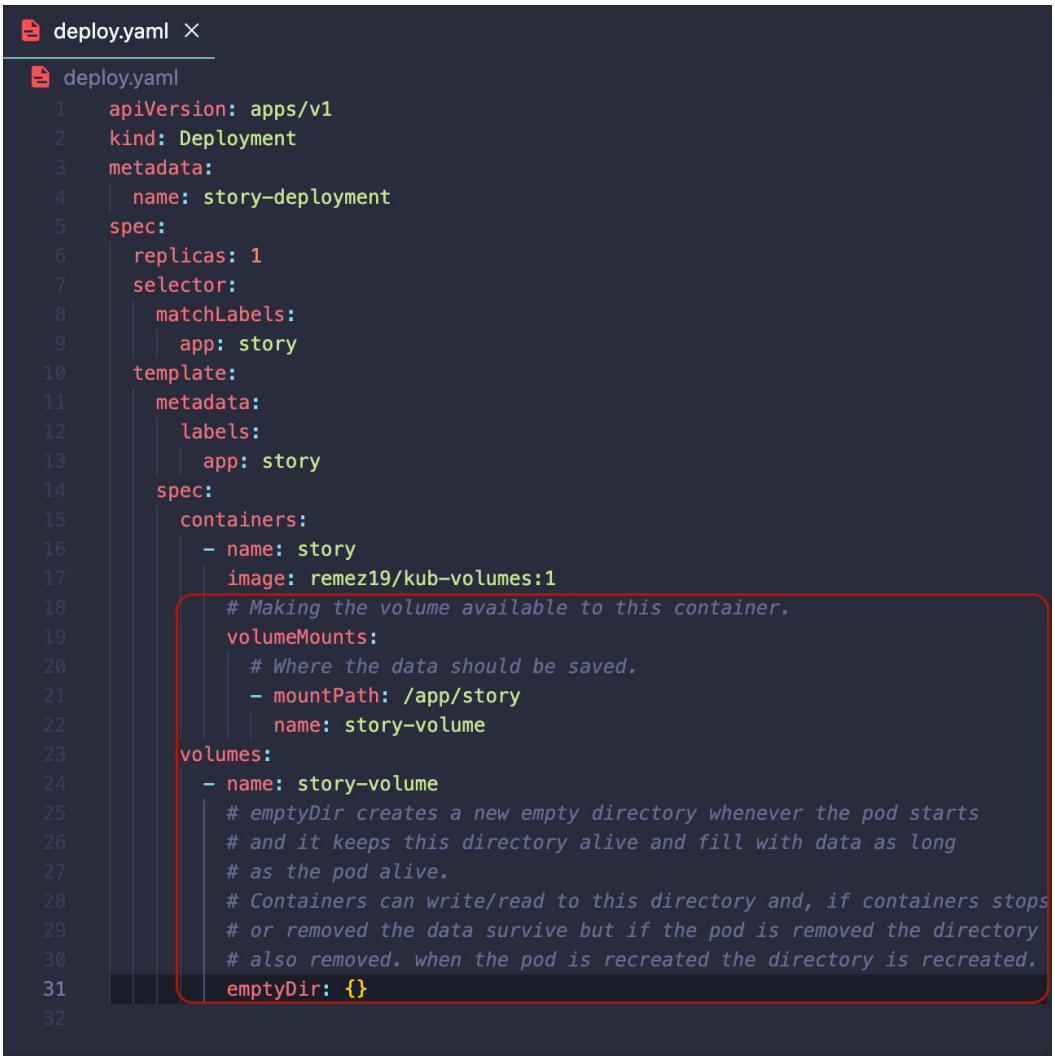


Creating a New Deployment & Service:

Getting Started with Kubernetes Volumes:

The official documentation - https://kubernetes.io/docs/concepts/storage/volumes/

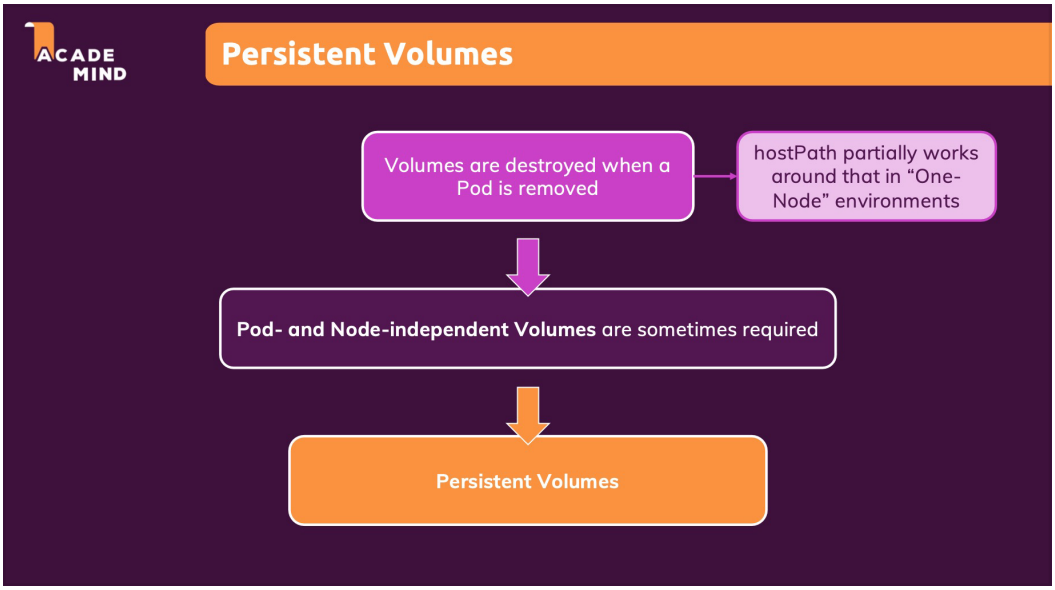
First Volume: The "emptyDir" Type:

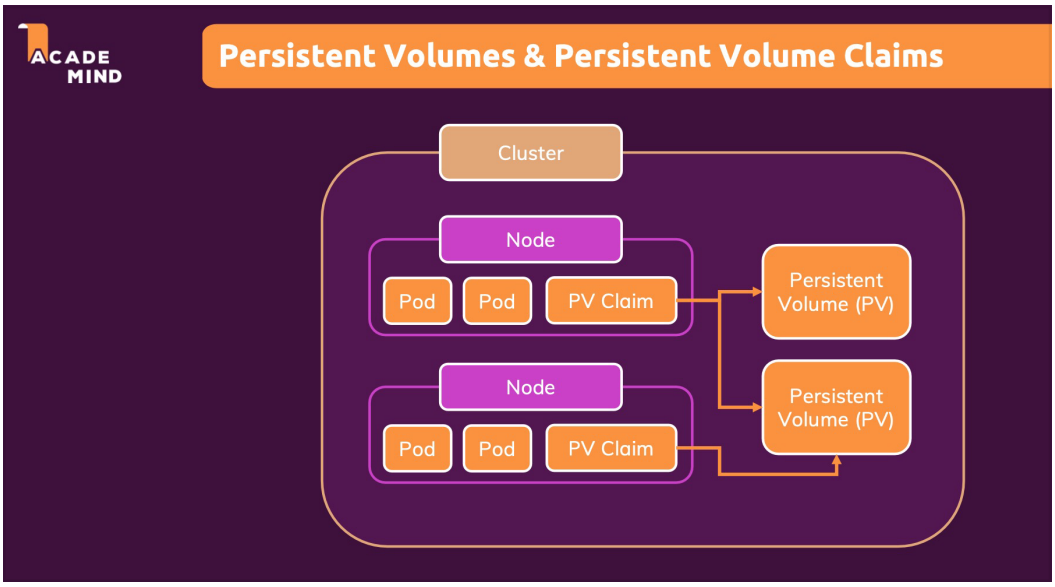


A Second Volume: The "hostPath" Type:

Understanding the "CSI" Volume Type:
The CSI volume type is a very flexible type which allows us to attach any storage solution out there in the world as long as there is an integration for this SCI type.

From Volumes to Persistent Volumes:
Up until now all the solutions we saw (volume types) have one disadvantage - the are destroyed when a pod is removed.
The hostPath works fine because we are using Minikube which contains only one Node (we only have one worker node).





Defining a Persistent Volume:

host-pv.yaml

```yaml
# Setting a Defining a Persistent Volume
# With the hostPath type/driver volume.
apiVersion: v1
kind: PersistentVolume
metadata:
  name: host-pv
spec:
  # The capacity is important because we as admins want to controll how much capacity
  # can be used by the diffrent pods.
  capacity:
    # The overall storage that can be used.
    # 4Gi - 4 Giga 4Gi
    storage: 1Gi
    # volumeMode - two optnios Filesystem/Block.
    # This are diffrent storage types.
    volumeMode: Filesystem
    # Define how this volume should be accessed.
    # We have three available options:
    # ReadWriteOnce - the volume can be accessed by only one node.
    # ReadOnlyMany - Rean only by many Nodes (cant work with hostPath - we have only
    # one Node).
    # ReadWriteMany
    accessMode:
      - ReadWriteOnce
  # This will setup a hosPath persistent volume as a stnd alone
  # detached from any pod in this enviroment. (only work when we have a single Node)
  hostPath:
    path: /data
    type: DirectoryOrCreate

```

Creating a Persistent Volume Claim:

This approach make sure that we have Persistent volume that is independent from the other nodes.
Persistent Volume

host-pvc.yaml deploy.yaml host-pv.yaml ×

host-pv.yaml

```yaml
# Setting a Defining a Persistent Volume
# With the hostPath type/driver volume.
apiVersion: v1
kind: PersistentVolume
metadata:
  name: host-pv
spec:
  # The capacity is important because we as admins want to controll how much capacity
  # can be used by the diffrent pods.
  capacity:
    # The overall storage that can be used.
    # 4Gi - 4 Giga 4Gi
    storage: 1Gi
    # volumeMode - two optnios Filesystem/Block.
    # This are diffrent storage types.
    volumeMode: Filesystem
    storageClassName: standard
    # Define how this volume should be accessed.
    # We have three available options:
    # ReadWriteOnce - the volume can be accessed by only one node.
    # ReadOnlyMany - Rean only by many Nodes (cant work with hostPath - we have only
    # one Node).
    # ReadWriteMany
    accessMode:
      - ReadWriteOnce
  # This will setup a hosPath persistent volume as a stnd alone
  # detached from any pod in this enviroment. (only work when we have a single Node)
  hostPath:
    path: /data
    type: DirectoryOrCreate

```

Claiming the persistent volume

host-pvc.yaml × deploy.yaml host-pv.yaml

host-pvc.yaml

```yaml
# The "Claim"
# 12:21

apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: host-pvc
spec:
  # Here we need to tell which persistent volume we want to claim
  # To connect the clain to the persistent volume we add the
  # volumeName: [VOLUME_NAME]
  volumeName: host-pv
  # we define how we want to aceess this volume.
  # we can have more then one way of accessing
  accessMode:
    - ReadWriteOnce
  # resources - is the counterpart to "capacity".
  storageClassName: standard
  resources:
    requests:
      storage: 1Gi # the maximum available for us.

```

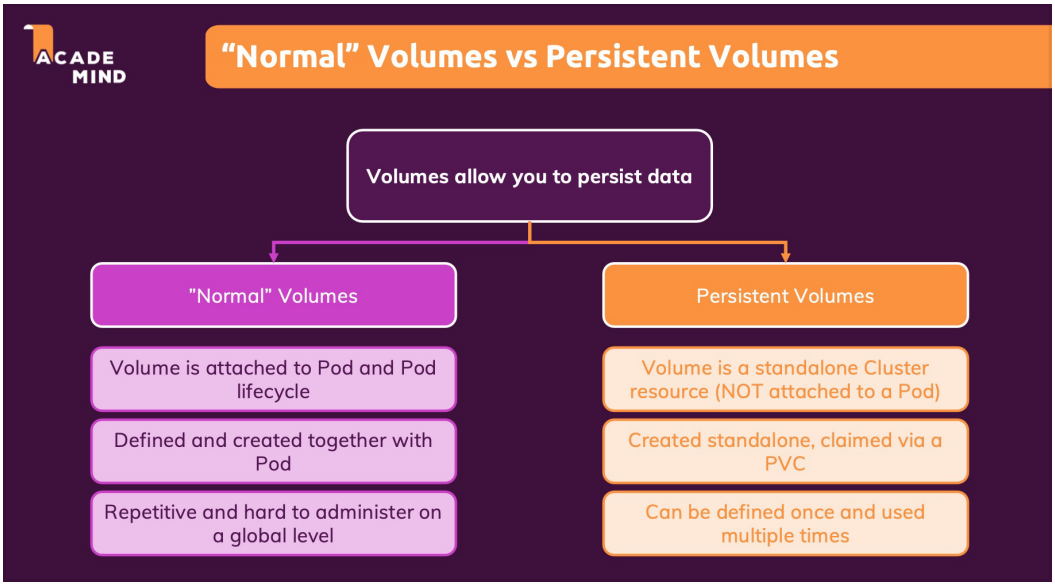
Connecting the pods to the claim we created.

host-pvc.yaml | deploy.yaml | host-pv.yaml

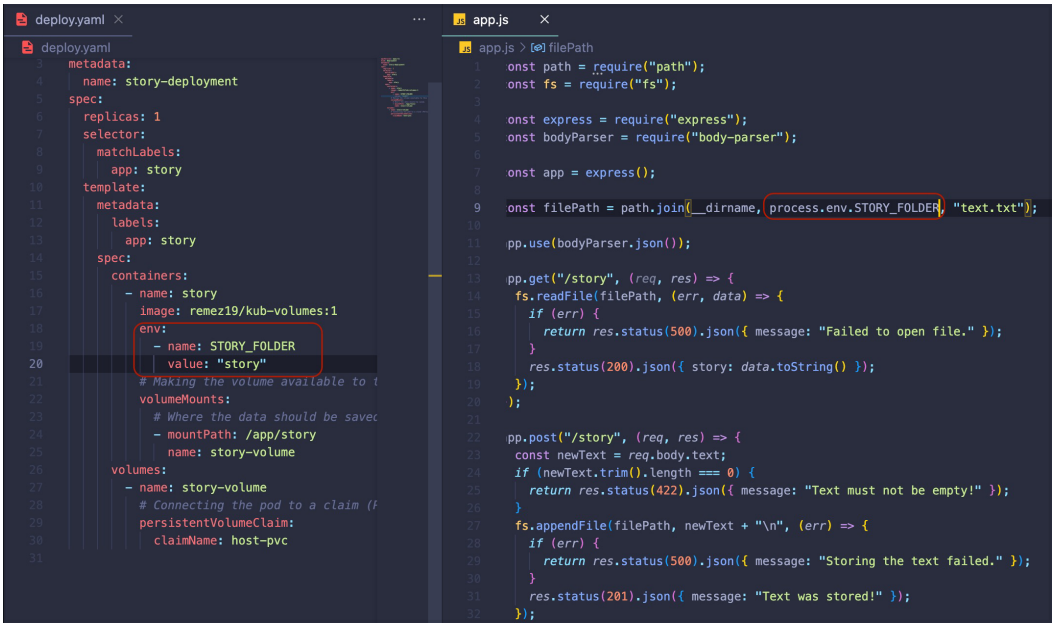
deploy.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: story-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: story
  template:
    metadata:
      labels:
        app: story
    spec:
      containers:
        - name: story
          image: remez19/kub-volumes:1
          # Making the volume available to this container.
          volumeMounts:
            # Where the data should be saved.
            - mountPath: /app/story
              name: story-volume
      volumes:
        - name: story-volume
          # Connecting the pod to a claim (Persistent Volume)
          persistentVolumeClaim:
            claimName: host-pvc
```

Volumes vs Persistent Volumes:

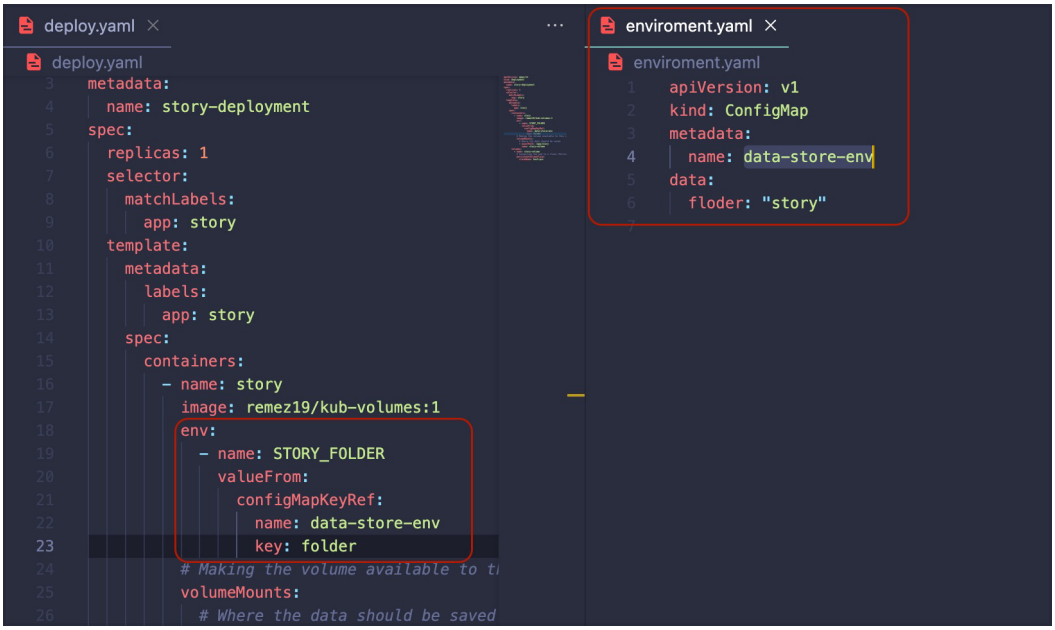


Using Environment Variables:



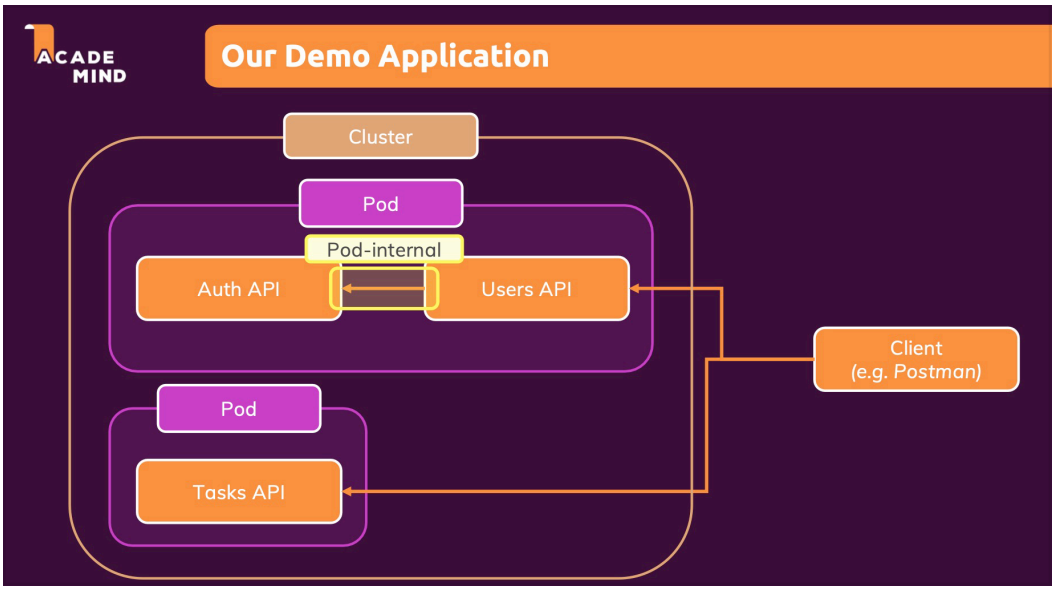
Environment Variables & ConfigMaps:

https://www.udemy.com/course/docker-kubernetes-the-practical-guide/learn/lecture/22627837#overview

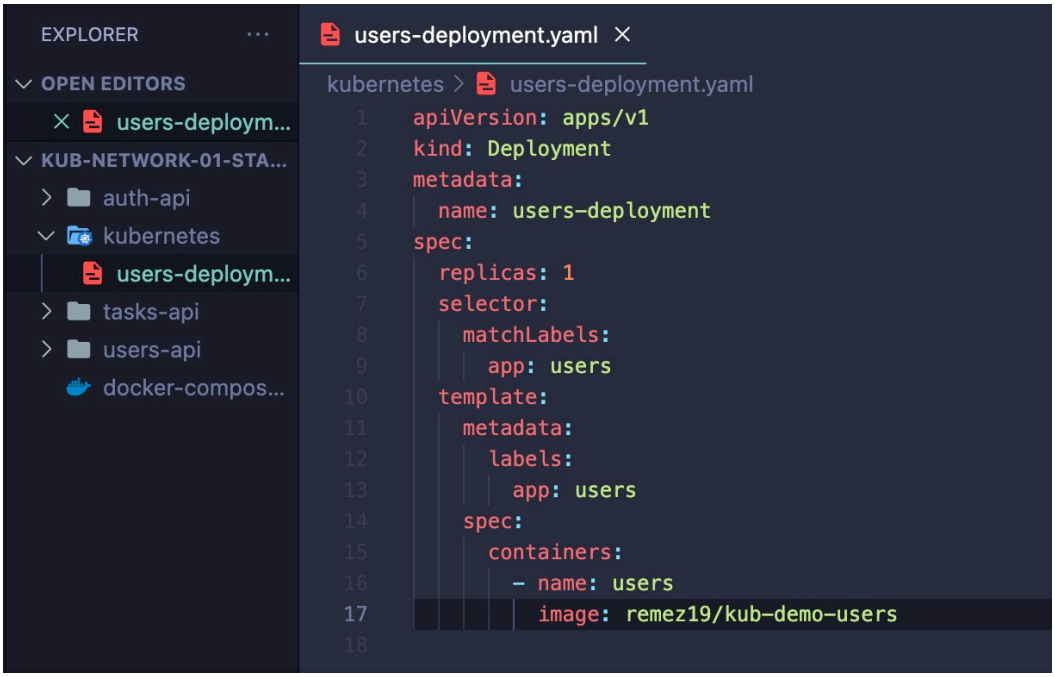


Section 14: Kubernetes Networking:

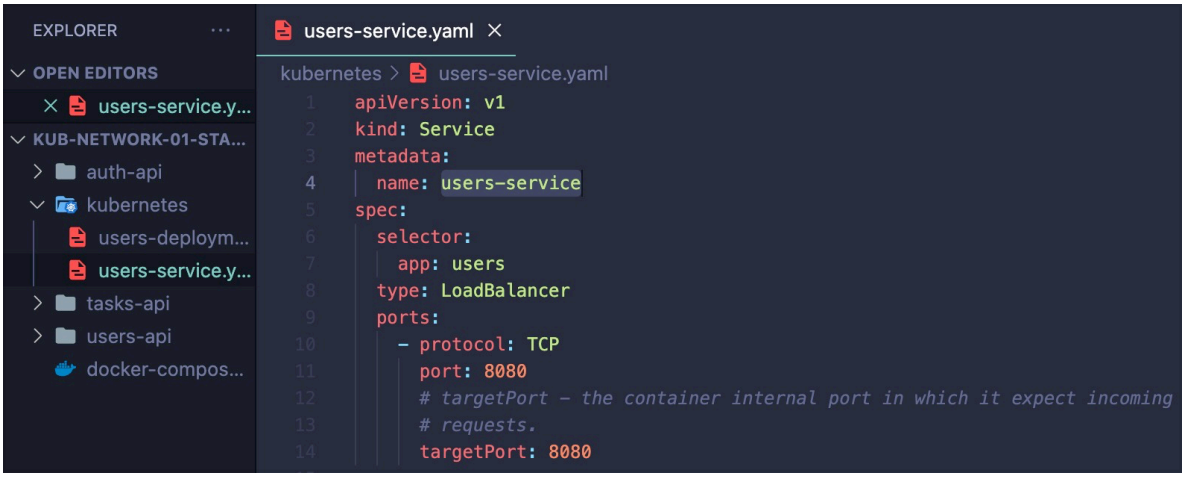
Starting Project & Our Goal:



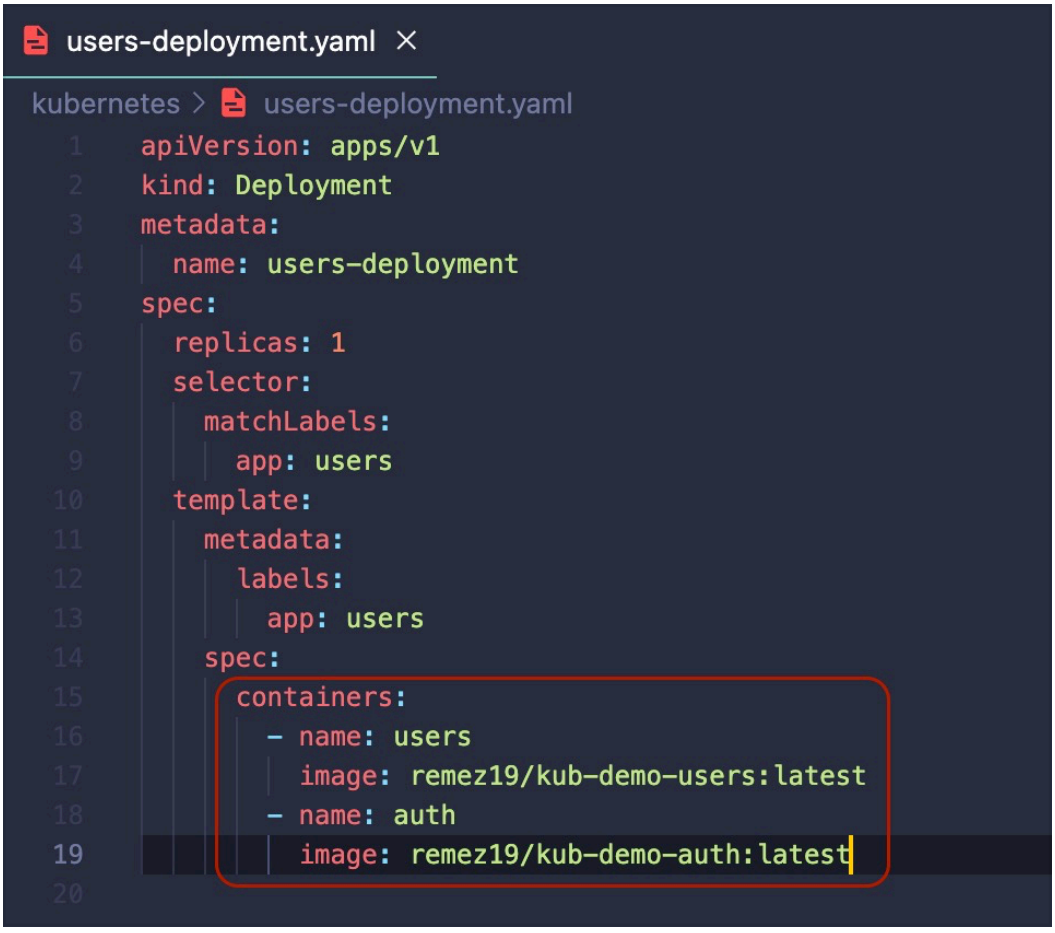
Creating a First Deployment:



Another Look at Services:



Multiple Containers in One Pod:



Pod-internal Communication:

Two containers in one pod communication.
Since we need one value in production and one value in development we use environment variables.

users-deployment.yaml

kubernetes > users-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: users-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      app: users
  template:
    metadata:
      labels:
        app: users
    spec:
      containers:
        - name: users
          image: remez19/kub-demo-users:latest
          env:
            - name: AUTH_ADDRESS
              value: localhost
        - name: auth
          image: remez19/kub-demo-auth:latest

```

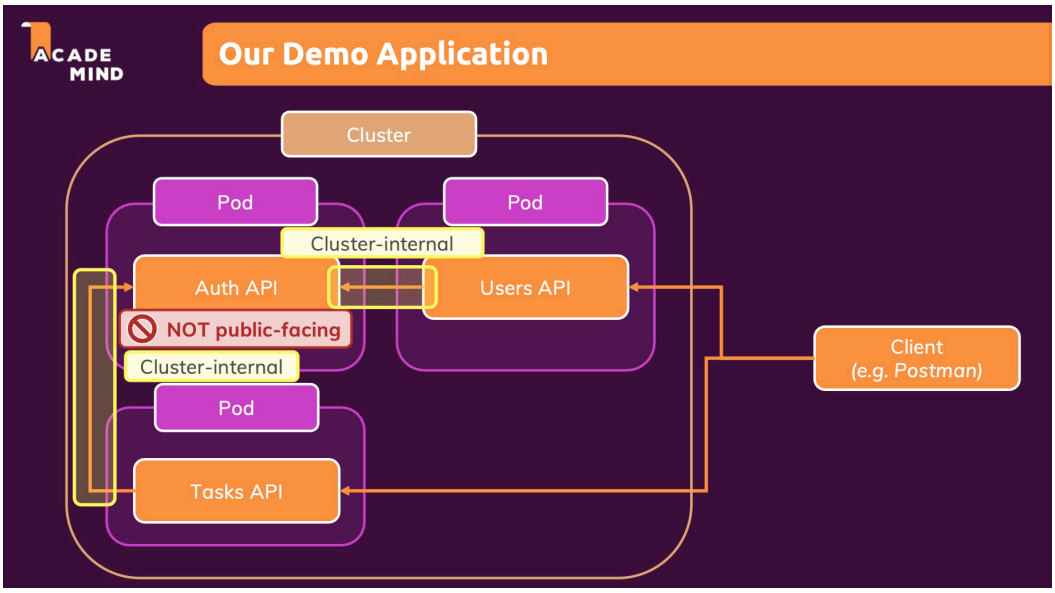
users-app.js

users-api > users-app.js > app.post("/login") callback > response

```
const express = require("express");
const bodyParser = require("body-parser");
const axios = require("axios");

const app = express();

app.use(bodyParser.json());

app.post("/signup", async (req, res) => {
  // It's just a dummy service - we don't really care for the email
  const email = req.body.email;
  const password = req.body.password;

  if (
    !password ||
    password.trim().length === 0 ||
    !email ||
    email.trim().length === 0
  ) {
    return res
      .status(422)
      .json({ message: "An email and password needs to be specified!" });
  }

  try {
    const hashedPW = await axios.get(
      `http://${process.env.AUTH_ADDRESS}/hashed-password/` + password
    );
    // const hashedPW = "dummy text";
    // since it's a dummy service, we don't really care for the hashed-pw either
    console.log(hashedPW, email);
    res.status(201).json({ message: "User created!" });
```

Creating Multiple Deployments:

We are going to look at pod-to-pod communication.



Pod-to-Pod Communication with IP Addresses & Environment Variables:

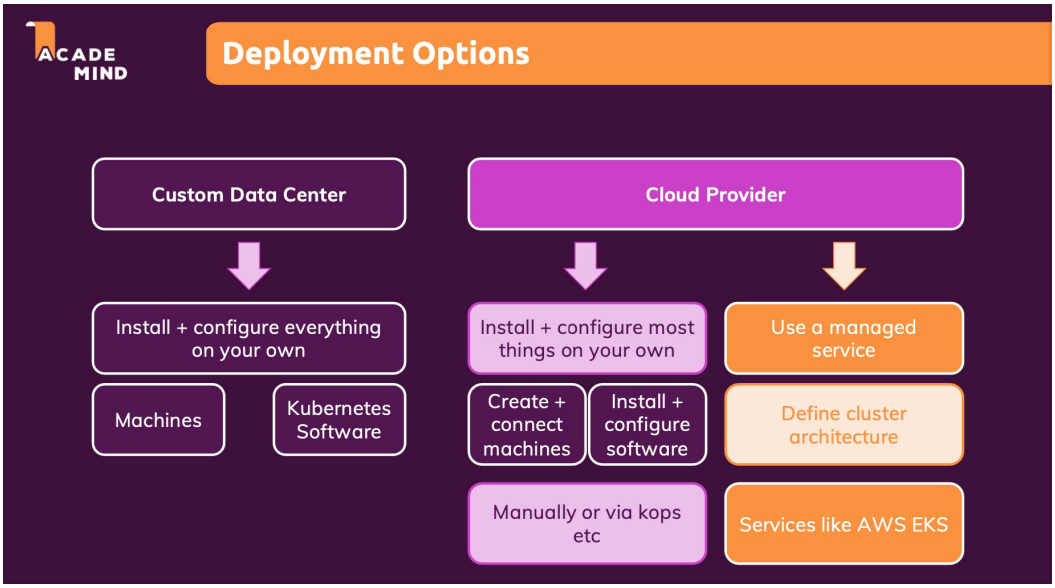
232:
https://www.udemy.com/course/docker-kubernetes-the-practical-guide/learn/

lecture/22627933#overview

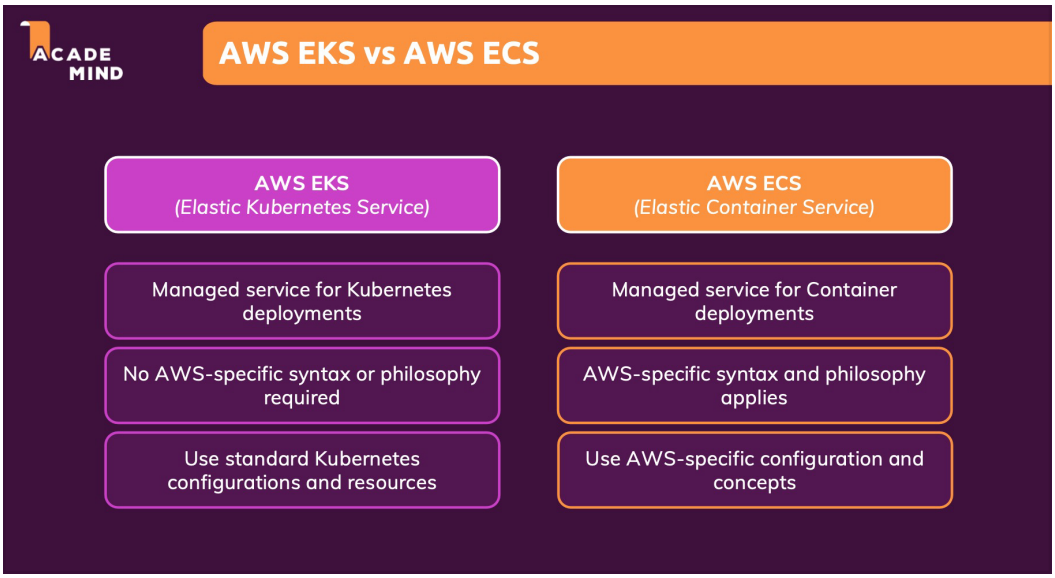
Using DNS for Pod-to-Pod Communication:
233
https://www.udemy.com/course/docker-kubernetes-the-practical-guide/learn/lecture/22627937#overview

Section 15: Kubernetes - Deployment (AWS EKS):

Deployment Options & Steps:



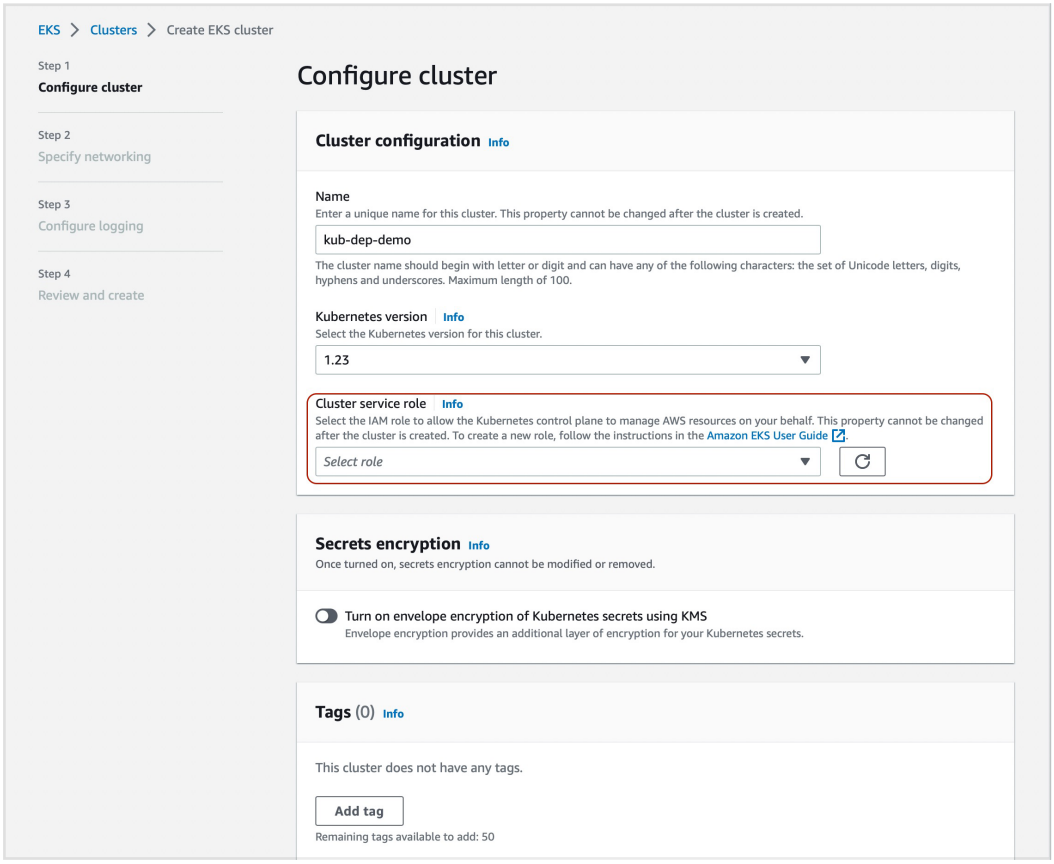
AWS EKS vs AWS ECS:

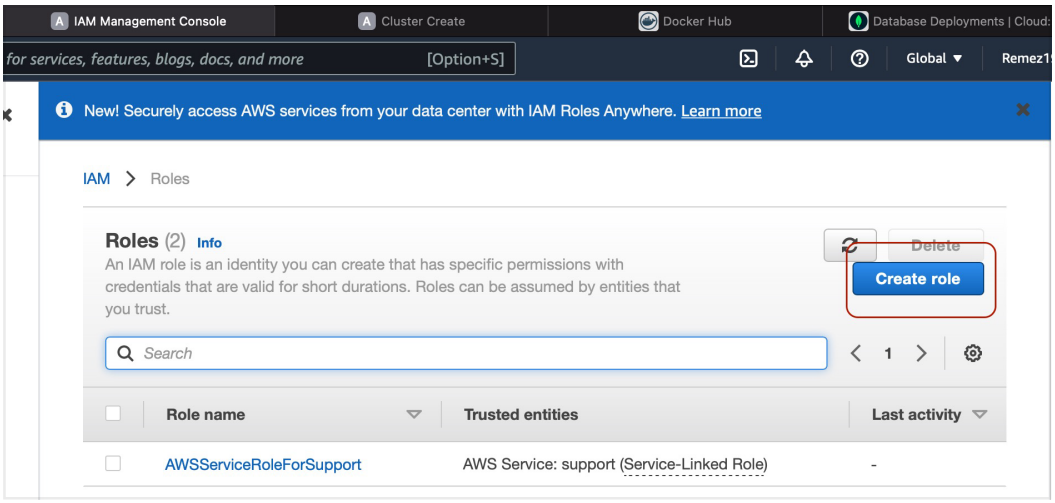


Creating & Configuring the Kubernetes Cluster with EKS:

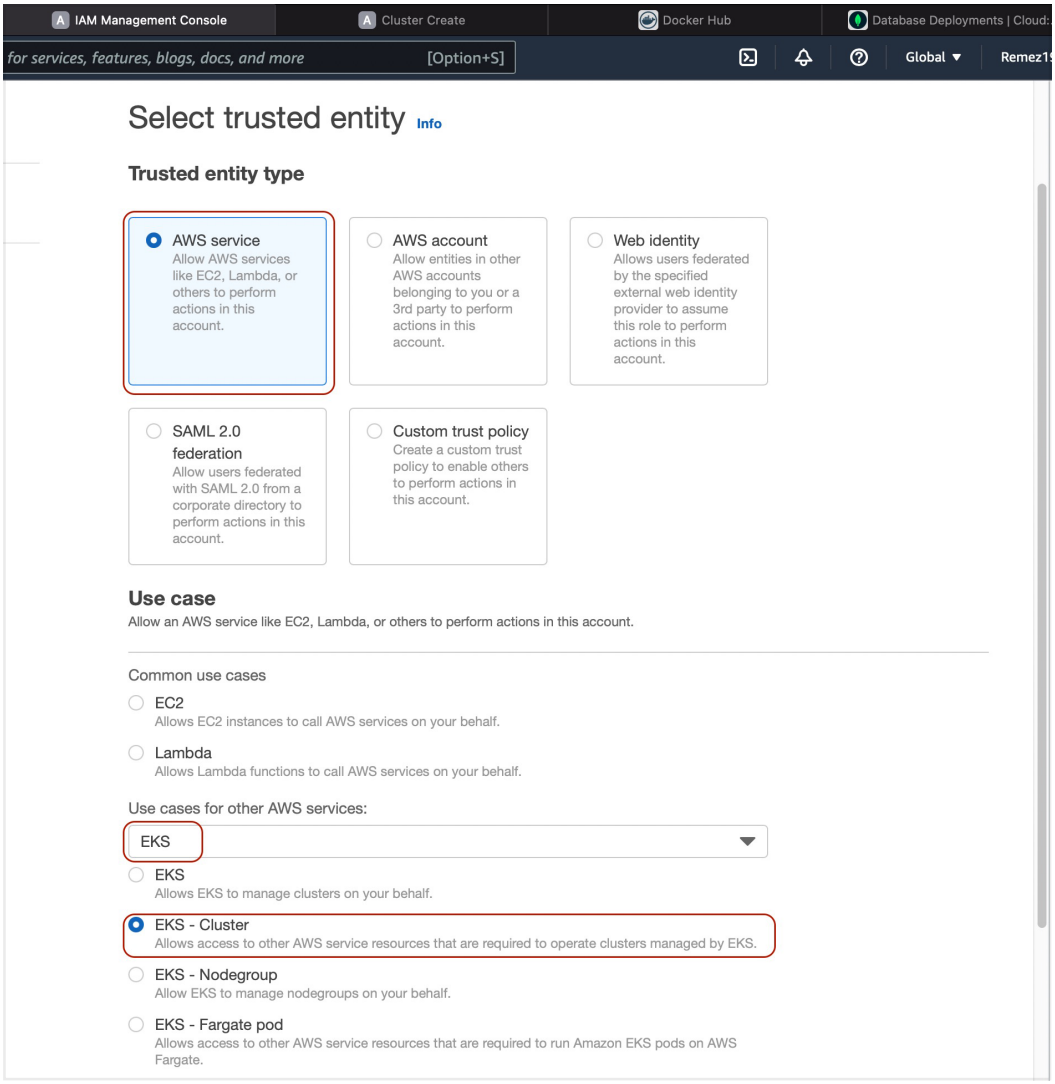
Cluster service role - in order to allow EKS to create other resources on our behalf we need to give the appropriate permissions.
We do that we that with another AWS service which allows us to manage

permissions - IAM (Identity Access Management) service.





EKS Cluster - this will give us a pre-defined role which is pre-defined for EKS to give EKS all the permissions it needs


IAM Management Console
Cluster Create
Docker Hub
Database Deployments | Cloud:
for services, features, blogs, docs, and more
[Option+S]
Global
Remez1
Select trusted entity Info
Trusted entity type
AWS service
Allow AWS services like EC2, Lambda, or others to perform actions in this account.
AWS account
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.
Web identity
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.
SAML 2.0 federation
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.
Custom trust policy
Create a custom trust policy to enable others to perform actions in this account.
Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.
Common use cases
EC2
Allows EC2 instances to call AWS services on your behalf.
Lambda
Allows Lambda functions to call AWS services on your behalf.
Use cases for other AWS services:
EKS
EKS
Allows EKS to manage clusters on your behalf.
EKS - Cluster
Allows access to other AWS service resources that are required to operate clusters managed by EKS.
EKS - Nodegroup
Allow EKS to manage nodegroups on your behalf.
EKS - Fargate pod
Allows access to other AWS service resources that are required to run Amazon EKS pods on AWS Fargate.