Docker Tutorial

1. Containers, Docker, Docker Compose
2. Docker Registry
3. Kubernetes
    a. Pods
    b. Workloads
    c. Services
    d. Updates
    e. Storage
    f. App Settings
    g. Observability
    h. Scaling

https://youtu.be/kTp5xUtcalw

https://docs.docker.com/engine/install/ubuntu/

https://landscape.cncf.io/

Why containers ?

- Move faster by deploying smaller units
- Use fewer resources
- Fit more into the same host
- Faster automation
- Portability
- Isolation

Container Registry

Orchestrator

- Manage
    - Infrastructure
    - Containers
    - Deployment
    - Scaling
    - Failover
    - Health monitoring
    - App upgrades, Zero-Downtime deployments
- Install your own
    - Kubernetes, Swarm, Service Fabric
- Orchestrators as a service
    - Azure Kubernetes Service, Service Fabric

What is Docker?

- An open source container runtime
- Mac, Windows & Linux support
- Command line tool
- Dockerfile file format for building container images

Docker commands

i. docker pull [imageName] - Pull an image from a registry
j. docker run [imageName] - Run containers
k. docker run -d [imageName] - detached mode
l. docker start [containerName] - start stopped containers
m. docker ps - list running containers
n. docker ps -a - list running and stopped containers
o. docker stop [containerName] - stop containers
p. docker kill [containerName] - kill containers
q. docker image inspect [imageName] - get image info
r. docker rm [containerName]
s. docker run -it nginx – /bin/bash
t. docker run -it – microsoft/powershell:nanoserver pwsh.exe →attach powershell
u. docker container exec -it [containerName] –bash → Attach to a running container
v. docker rm $(docker ps -a -q) → Removes all stopped containers
w. docker images → Lists images
x. docker rm [imageName]
y. docker system prune -a →Removes all images not in use by any containers

imageName - name of the image on the repo

docker run –publish 80:80 –name webserver nginxdo

Build containers

- docker build -t [name:tag] . → Builds an image using a Dockerfile located in the same folder
- docker build -t [name:tag] folder →Builds an image using a Dockerfile located in a different folder
- docker tag [imageName] [name:tag] →Tag an existing image
- **Example: Static HTML SITE**
    - FROM nginx:alpine
    - COPY . /usr/share/nginx/html
    - docker build -t webserver-image:v1 . → build
    - docker run -d -p 8080:80 webserver-image:v1 → run
    - Curl localhost:8000 → display

- **Example : Dockerfile - Node site**
    - FROM alpine
    - RUN apk add -update nodejs nodejs-npm
    - COPY . /src
    - WORKDIR /src
    - RUN npm install
    - EXPOSE 8080
    - ENTRYPOINT [“node”, “./app.js”]

Docker tagging

- docker tag → create a target image
    - name:tag
        - myimage:v1
    - repository/name:tag
    - myacr.azurecr.io/myimage:v1

Docker tutorial by Mosh

1. What is Docker

   A platform for building, running and shipping applications

2. What is a Container
    - allows running multiple apps in isolation
    - Are lightweight
    - Use OS of the host
    - Start quickly
    - Need less hardware resources
3. Docker Architecture
4. Docker Commands
    a. Build an image
        i. docker build -t hello-docker, `hello-docker` is a tag name
    b. List images
        i. docker image ls
        ii. docker images

Docker tutorial by **Academind**

1. Module 1 & 2
    a. Docker intro
        i. Image vs Container
            1. Images - templates/ blueprints for containers
            2. Containers - the running unit of software
        ii. Images
            1. Images are layer based and each layer is cached. Each instruction corresponds to a layer.
            2. Images are read only

    b. Docker commands
        i. Build an image
            1. docker build [path to the Dockerfile]
            2. docker build -t [nameOfTheImage] .
        ii. Managing Images & Containers
            1. Running the containers
                a. docker run -p 3000:80 [imageName/Id]
                b. docker run -p 3000:80 -d [imageName/Id] →running in a detached mode
                c. docker run -p 3000:80 -d --rm [imageName/Id] →running in a detached mode and removes the container when exited
            2. List containers
                a. docker ps
                b. docker ps -a → list stopped containers too
            3. Restart a stopped container
                a. docker start [containerName]
                b. docker start -a [containerName] → in attached mode
            4. Attach to a container
                a. docker attach [containerName/id]
            5. View the logs
                a. docker logs →view all the logs
                b. docker logs -f → similar to tail -f, you can see the logs in realtime
            6. Attach in interactive mode
                a. docker run -it [imageName/id]
                b. docker start -a -i [containerName]
            7. Deleting images & Containers
                a. docker rm [containerName] [containerName]
                b. docker rmi [imageName]
                c. docker image prune → remove unused containers

8. Inspecting the image
    a. docker image inspect [imageName/id]
9. Copying files into and from a container
    a. From local machine to container
        i. docker cp dummy/. jolly_carson:/test → docker cp [sourceFolder/filePath] [containerName]:[pathInsideTheContainer]
    b. From container to a local machine
        i. docker cp jolly_carson:/test/test.txt dummy/ →docker cp [containername]:[pathInsideTheContainer] [destinationPath]
10. Naming containers and images
    a. Naming a container
        i. docker run -p 3000:80 -d --rm –name [anyNameForTheContainer] [imageName/Id]
    b. Naming/tagging an image
        i. docker build -t [anyName]:[tag/version] .
        ii. Renaming
            1. docker tag node-hello-world:latest mahatoniteesh/node-hello-world
11. Pushing and pulling images to/from docker hub
    a. docker login → login to the hub
    b. docker push [imageName] → image name should be same as the repository name create online e.g. mahatoniteesh/node-app
    c. docker pull [repositoryName]:[version]

2. Module 3
    a. Volumes
        i. Are folder that reside on the host machine
        ii. These are used by container to store files
        iii. When the container is killed/stopped, the data persists
        iv. Types:
            1. Anonymous Volume
                a. docker run -v /app/data…
            2. Name volume
                a. docker run -v data:/app/data..
            3. Bind Mount
                a. docker run -v /path/to/code:/app/code…
    b. Named Volumes
        i. Creating name volume
            1. docker run --name feedback-app -d -p 3000:80 -v feedback:/app/feedback feedback:volume → -v flag is used to

attach a name volume to the container which is persisted when the container gets terminated

2. -v [volumeName]:[pathInsideTheContainerToLink]

ii. Listing the volumes
   1. docker volume ls

iii. Removing the volumes
   1. docker volume rm [volumeName]
   2. docker volume prune

c. Bind Mounts (Code sharing)
   i. Creating
      1. docker volume create [volumeName]
      2. -v $(pwd):/app → create while container run command
   ii. Running container with bind mount
      1. docker run --name feedback-app -d -p 3000:80 -v feedback:/app/feedback -v $(pwd):/app -v /app/node_modules feedback:volume
   iii. Read only volume
      1. -v feedback:/app/feedback -v $(pwd):/app:ro → add `ro`

d. ENV variables
   i. Provide env
      1. In docker file
         ENV PORT 80
         EXPOSE $PORT
      2. -e PORT=3000
   ii. Provide env file
      1. –env-file .env
   iii. While running container
      1. docker run --name feedback-app -e PORT=8000 -p 3000:8000 -d -v $(pwd):/app -v /app/node_modules -v feedback:/app/feedback --rm feeback-app:env

e. ARG variables
   i. In docker file
      1. ARG DEFAULT_PORT=80
         ENV PORT $DEFAULT_PORT
      2. docker build –build-arg DEFAULT_PORT=8000

3. Module 4 - Networking
   a. Accessing localhost
      i. host.docker.internal → localhost on the host machine
   b. Container to Container
      i. Using container ip address → inefficient
      ii. Using docker network
         1. docker network create [networkName]
      iii. Adding container to a network

            1. docker run –network [networkName] → you can connect to other containers using the containerName http://[containerName]

4. Module 5 - Multi Container apps
5. Module 6 - Docker Compose
    a. Installing
        i. On macOS and Windows, you should already have Docker Compose installed - it's set up together with Docker there.
        ii. On Linux machines, you need to install it separately.
        iii. These steps should get you there:
        iv. 1. `sudo curl -L "https://github.com/docker/compose/releases/download/1.27.4/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose`
        v. 2. `sudo chmod +x /usr/local/bin/docker-compose`
        vi. 3. `sudo ln -s /usr/local/bin/docker-compose /usr/bin/docker-compose`
        vii. 4. to verify: `docker-compose --version`
        viii. Also see: https://docs.docker.com/compose/install/

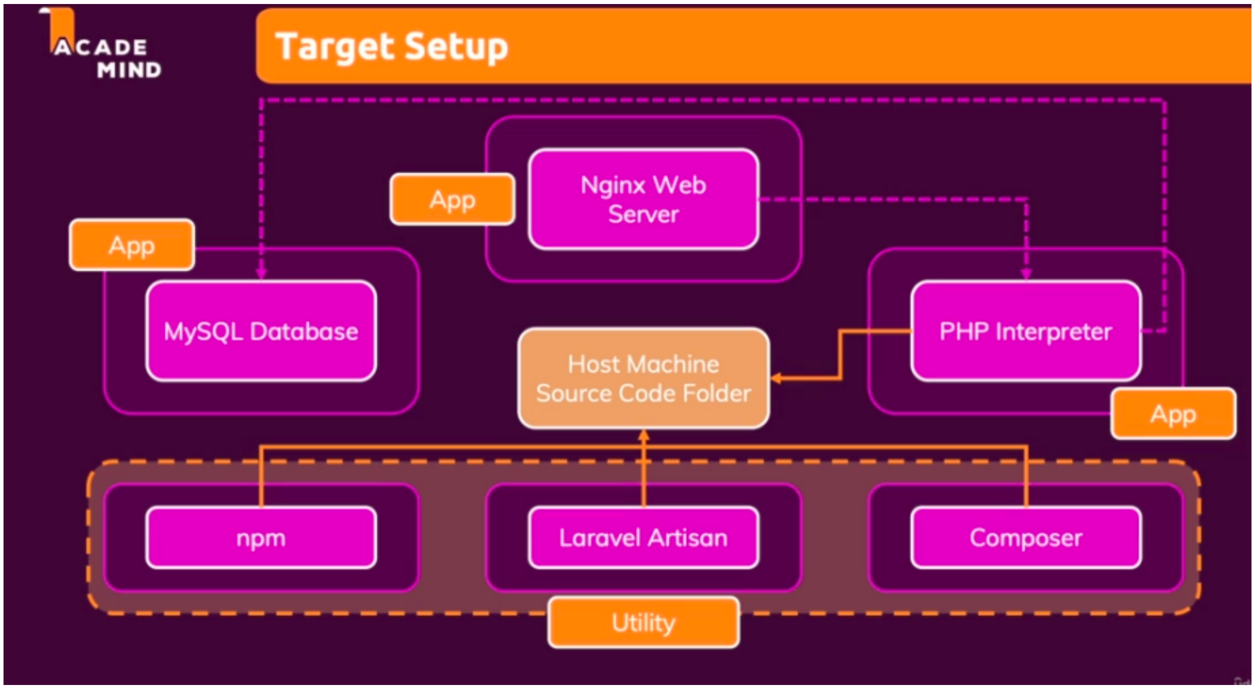
    b. Writing docker compose file

```
version: "3.8"
services:
 mongodb:
   image: "mongo"
   volumes:
     - ./mongo-data:/data/db
   # environment:
   #   - MONGO_INITDB_ROOT_USERNAME=$MONGO_USER
   #   - MONGO_INITDB_ROOT_PASSWORD=$MONGO_PASS
   env_file:
     - ./env/mongo.env
   # networks:
   #   - network
   # container_name: mongodb
 backend:
   # build: path
   # build:
```

```
   #   context: path
   #   dockerfile: Dockerfile
   #   args:
   #     - name=value
   ports:
     - "80:3000"
   volumes:
     - m4backend-logs:/app/logs
     - ./backend:app
     - /app/node_modules
   env_file:
     - ./env/backend.env
   depends_on:
     - mongodb
 frontend:
   build: ./frontend
   ports:
     - '3000:3000'
   volumes:
     - ./frontend/src:app/src
   stdin_open: true
   tty: true
   depends_on:
     - backend
volumes:
 m4backend:
```

6. Module 7 - utility containers
    a. docker exec -it <container-name> npm init → to execute a command inside the container
    b. docker run -it <image-name> <command-to-run> → docker run -it node npm init
    c. You can specify entrypoint to restrict the command
        i.
        ```
        ENTRYPOINT ["npm"]
        ```
        ii. docker run -t node init → which will be then resolved to npm init
        iii. For compose file
            1. docker compose run <container-name/service-name> init

7. Module 8 – Laravel php setup



commands :

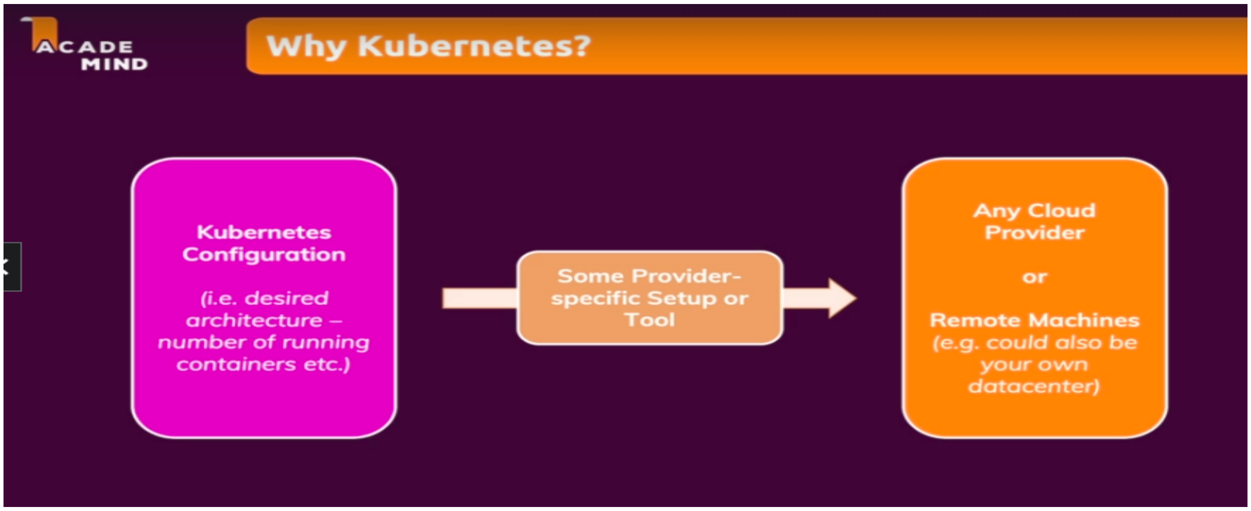
- Docker-compose up -d –build <service-name> → force build containers

8. Module 9 – Deploying containers

Check cloud formation
Cloudwatch

Kubernetes

1. Introduction
   a. Why Kubernetes ?



ACADE MIND

Why Kubernetes?

Kubernetes Configuration

(i.e. desired architecture – number of running containers etc.)

Some Provider-specific Setup or Tool

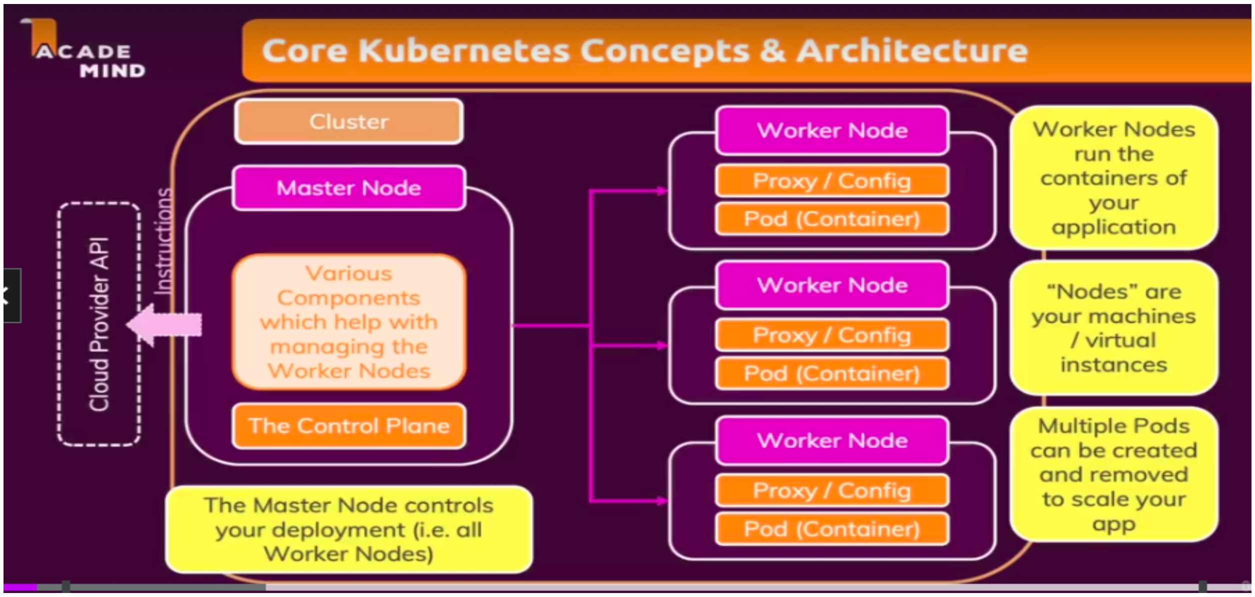
Any Cloud Provider

or

Remote Machines (e.g. could also be your own datacenter)

b. Concepts
   i. Pod → Smallest possible unit
      1. Contains 1 or more containers
   ii. Worker node → A virtual machine eg. ec2
      1. On which 1 or more pods run
   iii. Proxy/config
      1. Used for networking.
   iv. Master Node → Another machine
      1. Controls worker node

All the above together forms the cluster



ACADE MIND

Core Kubernetes Concepts & Architecture

Cluster

Master Node

Various Components which help with managing the Worker Nodes

The Control Plane

Instructions

Cloud Provider API

The Master Node controls your deployment (i.e. all Worker Nodes)

Worker Node

Proxy / Config

Pod (Container)

Worker Node

Proxy / Config

Pod (Container)

Worker Node

Proxy / Config

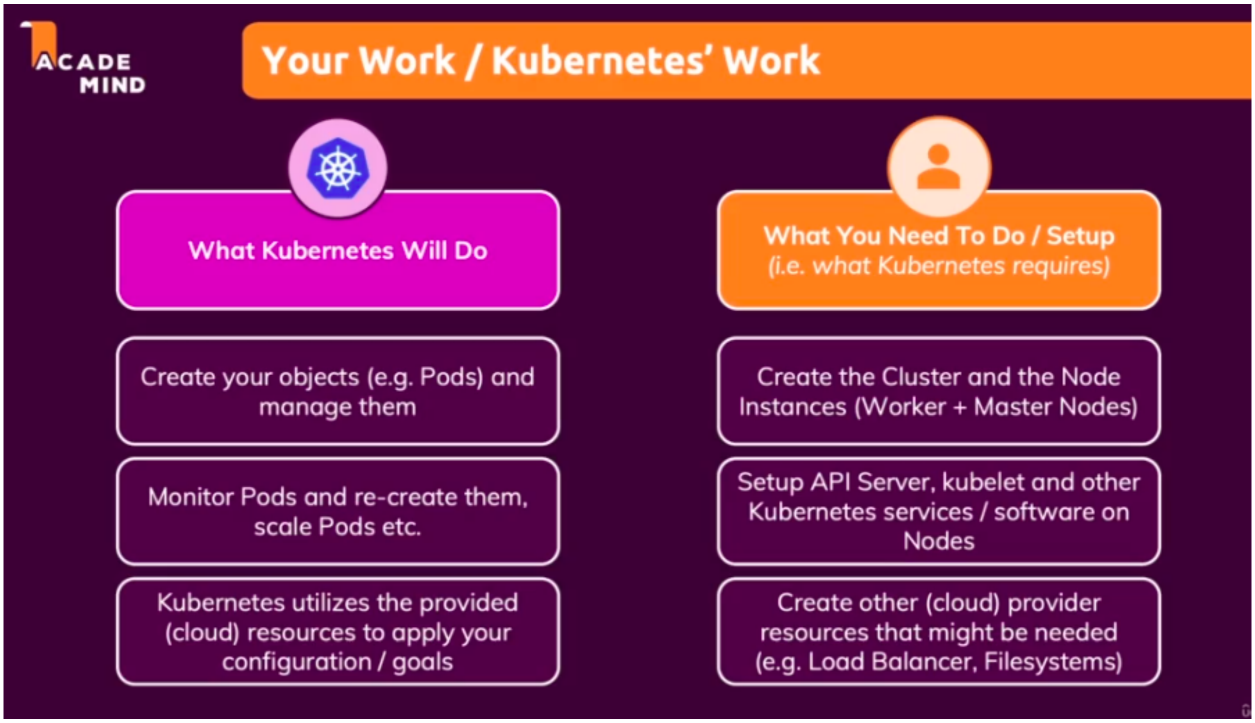
Pod (Container)

Worker Nodes run the containers of your application

"Nodes" are your machines / virtual instances

Multiple Pods can be created and removed to scale your app

Your Work / Kubernete Work



ACADEMIND

Your Work / Kubernetes' Work

What Kubernetes Will Do

Create your objects (e.g. Pods) and manage them

Monitor Pods and re-create them, scale Pods etc.

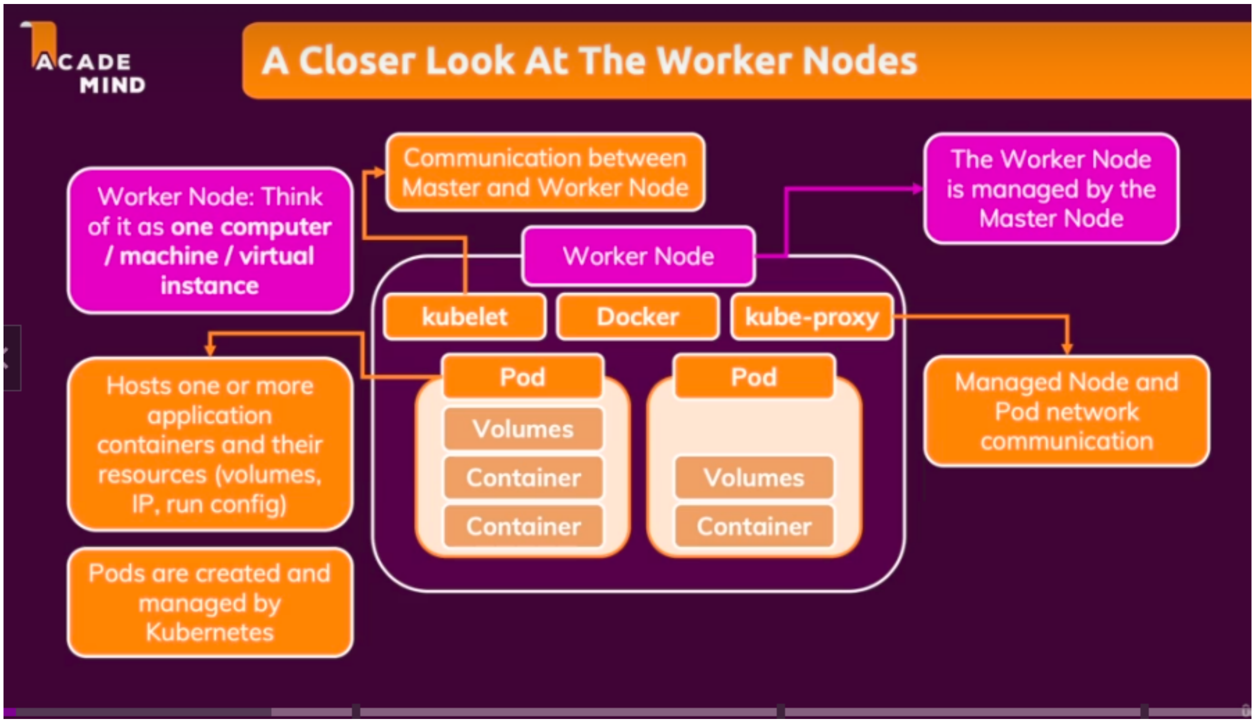
Kubernetes utilizes the provided (cloud) resources to apply your configuration / goals

What You Need To Do / Setup
(i.e. what Kubernetes requires)

Create the Cluster and the Node Instances (Worker + Master Nodes)

Setup API Server, kubelet and other Kubernetes services / software on Nodes

Create other (cloud) provider resources that might be needed (e.g. Load Balancer, Filesystems)



ACADEMIND

A Closer Look At The Worker Nodes

Worker Node: Think of it as **one computer / machine / virtual instance**

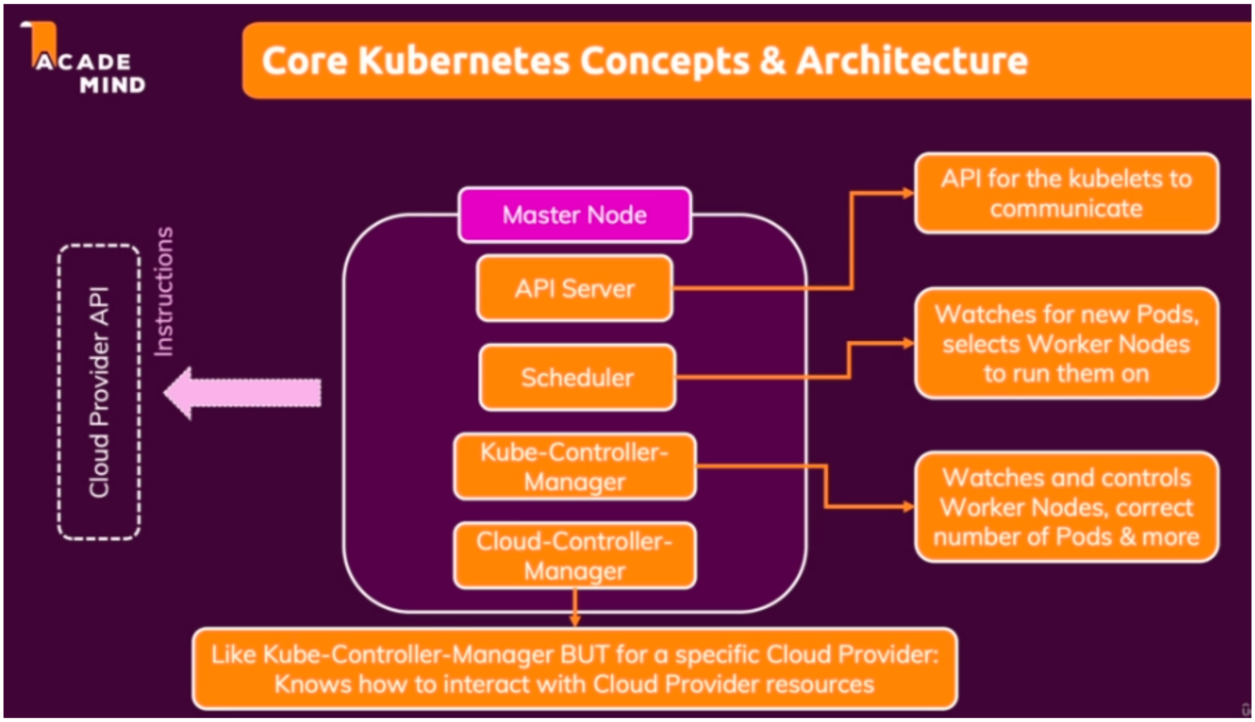
Communication between Master and Worker Node

The Worker Node is managed by the Master Node

Hosts one or more application containers and their resources (volumes, IP, run config)

Managed Node and Pod network communication

Pods are created and managed by Kubernetes



Core Kubernetes Concepts & Architecture

Master Node

API Server

Scheduler

Kube-Controller-Manager

Cloud-Controller-Manager

API for the kubelets to communicate

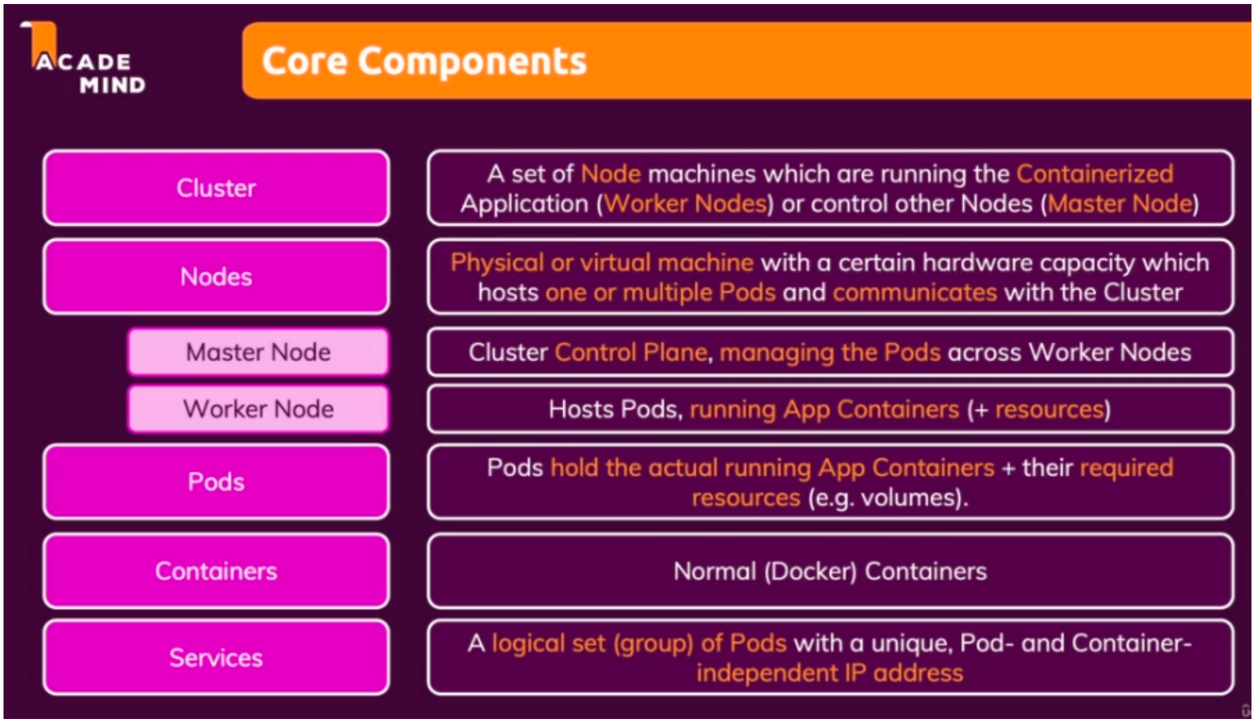
Watches for new Pods, selects Worker Nodes to run them on

Watches and controls Worker Nodes, correct number of Pods & more

Like Kube-Controller-Manager BUT for a specific Cloud Provider: Knows how to interact with Cloud Provider resources

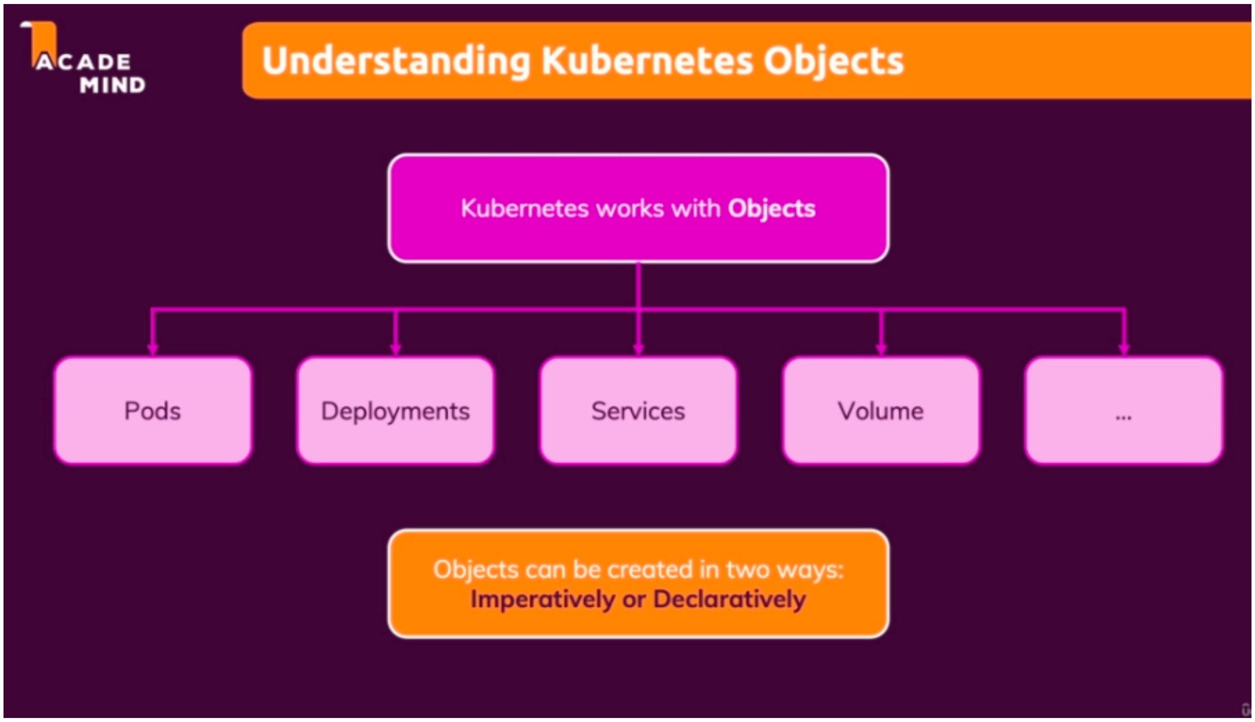
Cloud Provider API

Instructions



Core Components

Cluster	A set of Node machines which are running the Containerized Application (Worker Nodes) or control other Nodes (Master Node)
Nodes	Physical or virtual machine with a certain hardware capacity which hosts one or multiple Pods and communicates with the Cluster
Master Node	Cluster Control Plane, managing the Pods across Worker Nodes
Worker Node	Hosts Pods, running App Containers (+ resources)
Pods	Pods hold the actual running App Containers + their required resources (e.g. volumes).
Containers	Normal (Docker) Containers
Services	A logical set (group) of Pods with a unique, Pod- and Container-independent IP address



ACADE MIND

Understanding Kubernetes Objects

Kubernetes works with **Objects**

Pods

Deployments

Services

Volume

...

Objects can be created in two ways:
Imperatively or Declaratively

2. Commands

a. Deployment Object

   i. Get deployments

      1. kubectl get deployments

   ii. Get pods

      1. kubectl get pods

   iii. Delete deployments

      1. kubectl delete deployment <app-name>

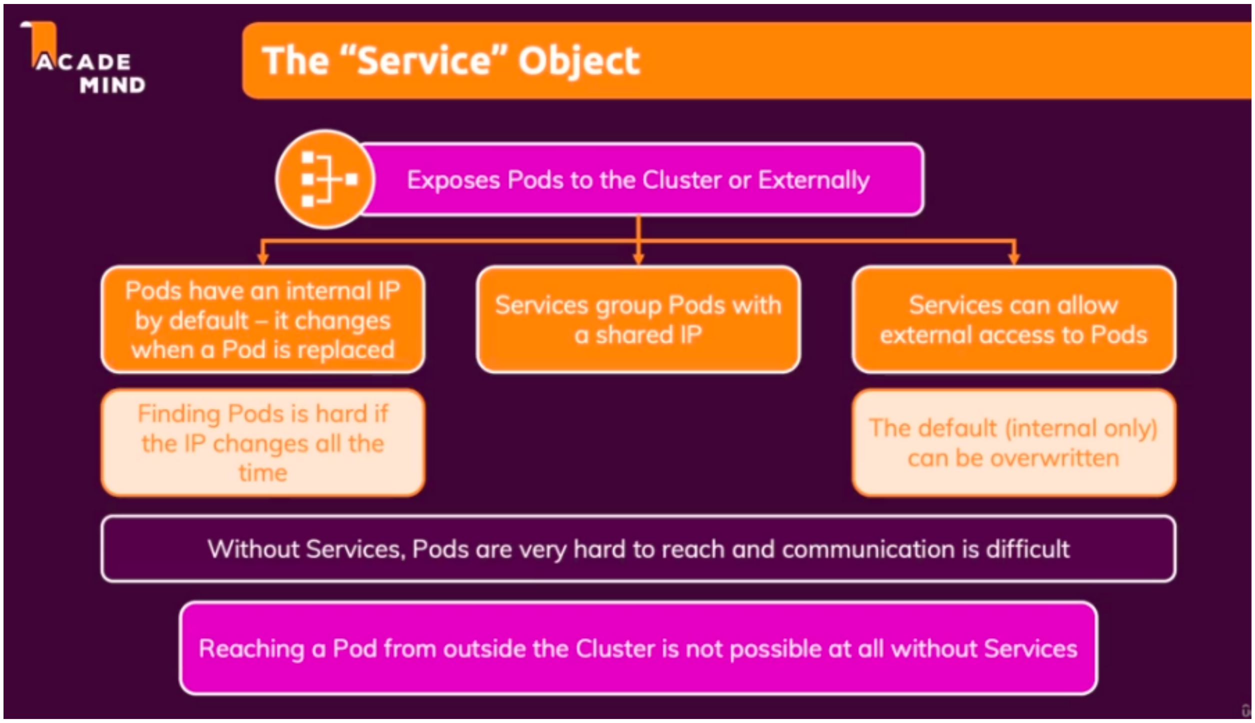
   iv. Create deployment

      1. kubectl create deployment first-app --image=<docker-hub-repo-name>

   v. Web dashboard

      1. minikube dashboard

b. Service Object → Expose pods to the cluster or Externally.



ACADE MIND

The "Service" Object

Exposes Pods to the Cluster or Externally

Pods have an internal IP by default – it changes when a Pod is replaced

Finding Pods is hard if the IP changes all the time

Services group Pods with a shared IP

Services can allow external access to Pods

The default (internal only) can be overwritten

Without Services, Pods are very hard to reach and communication is difficult

Reaching a Pod from outside the Cluster is not possible at all without Services

- i. Creating a service
  1. kubectl service create
- ii. Expose a port
  1. kubectl expose deployment <app-name> –type=<type-of-expose> –port=8080
     - a. Expose types can be
       - i. ClusterIP → will give an ip that is reachable only inside the cluster
       - ii. NodePort →will give an ip of the the worker node, now the app will be accessible to the outside world
       - iii. LoadBalancer →this will assign a public ip and will also distribute the load
  2. Locally assigning ip to a loadbalancer service
     - a. minicube service <app-name>
- iii. List services
  1. kubectl get services
- iv. Scaling - create replicas
  1. kubectl scale deployment/first-app --replicas=3 → will create 3 replica pods of deployment/first-app
  2. kubectl scale deployment/first-app --replicas=1 → will downgrade to 1 replica pod of deployment/first-app

c. Updating deployments

   i. updating an image

      1. kubectl set image deployment/first-app kub-first-app=mahatoniteesh/kub-first-app

   ii. Get rollout status/update status

      1. kubectl rollout status deployment/first-app

   iii. Get rollout history of a deployment

      1. kubectl rollout history deployment/first-app

   iv. Rolling back to a particular revision

      1. kubectl rollout undo deployment/app –to-revision=2 → rolling back to the second revision

d. Config files

   i. Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
 name: second-app-deployment
spec:
 replicas: 1
 selector:
   matchLabels:
     app : second-app
     tier : backend
 template:
   metadata:
     labels:
       app: second-app
       tier: backend
   spec:
     containers:
       - name: second-node
         image: mahatoniteesh/kube-first-app
```

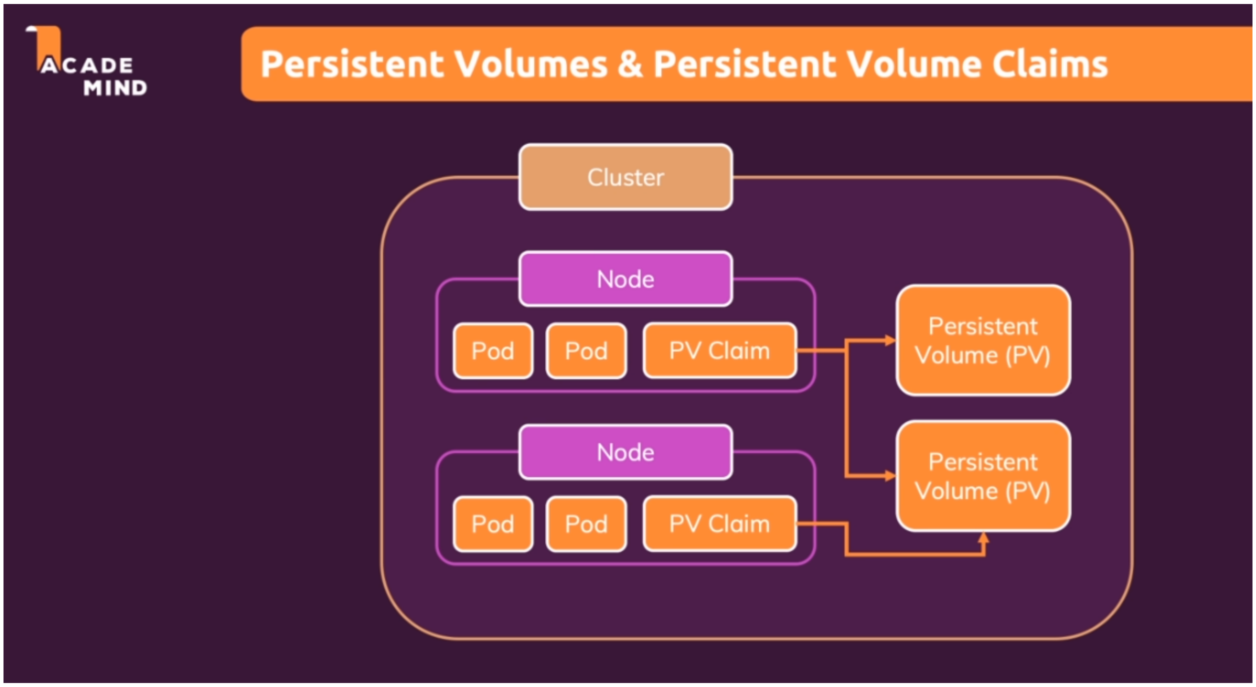
   ii. Service

```
apiVersion: v1
kind: Service
metadata:
 name: backend
spec:
 selector:
   app: second-app
```

```
ports:
  - protocol : 'TCP'
    port: 80
    targetPort: 8080
type: LoadBalancer
```

iii. Commands
   1. Starting from a file
      a. kubectl apply -f=deployment.yaml
   2. Delete the resources
      a. kubectl delete -f=deployment.yaml -f=... -f=..
   3. Delete by label
      a. kubectl delete deployments,services -l group=example

e. Volume
   i. emptyDir
   ii. hostPath
   iii. CSI
   iv. Persistent Volume
      1. Get
         a. kubectl get pv
      2. Get storage class
         a. kubectl get sc
      3. Get storage claims
         a. kubectl get pvc



Persistent Volumes & Persistent Volume Claims

Persistent Volume

```
apiVersion: v1
```

```
kind: PersistentVolume
metadata:
 name: story-pv-volume
spec:
 storageClassName: manual
 capacity:
   storage: 1Gi
 volumeMode: Filesystem
 storageClassName: standard
 accessModes:
   - ReadWriteOnce
 hostPath:
   path: "/data"
   type: DirectoryOrCreate
```

Persistent volume claim

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
 name: story-pvc
spec:
 volumeName: story-pv-volume
 storageClassName: standard
 accessModes:
   - ReadWriteOnce
 resources:
   requests:
     storage: 1Gi
```

Deployment

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

```
metadata:
  labels:
    app: story
spec:
  containers:
    - name: story
      image: mahatoniteesh/kub-volumes-demo
      volumeMounts:
        - mountPath: /app/story
          name: story-volume
      env:
        - name: STORY_FOLDER
          value: 'story'
  volumes:
    # - name: story-volume
    #   emptyDir: {}
    # - name: story-volume
    #   hostPath:
    #     path: /data
    #     type: DirectoryOrCreate
    - name: story-volume
      persistentVolumeClaim:
        claimName: story-pvc
```

v. Environment variables and configmap

```
spec:
  containers:
    - name: story
      image: mahatoniteesh/kub-volumes-demo
      volumeMounts:
        - mountPath: /app/story
          name: story-volume
      env:
        - name: STORY_FOLDER
          value: 'story'
      env:
        - name: STORY_FOLDER
          valueFrom:
            configMapKeyRef:
              name: data-store-env
            key: folder
```

vi. Config maps

```
apiVersion: v1
kind: ConfigMap
metadata:
 name: data-store-env
data:
 folder: 'story'
```

f. Networking

  i. To communicate with a container within a pod
    1. Use `localhost`
  ii. To communicate between pods
    1. Create a service for each pod
    2. Connect
      a. Method 1
        i. User env var generated by kubernetes
          1. It will be in the form, suppose your servce name is **auth-service** then the env variable will be **AUTH_SERVICE_SERVICE_HOST**. So depending on your service name the first part will change but **_SERVICE_HOST** will remain the same.
        ii. Using internal cluster DNS by kubernates
          1. They are in the form **`<service_name>.<namespace>**
          2. So if the service name is **auth-service** and the namespace is **default** then the dns will be **auth-service.default**
          3. You can get the namespace name by **kubectl get namespaces**

g. Deployment aws eks