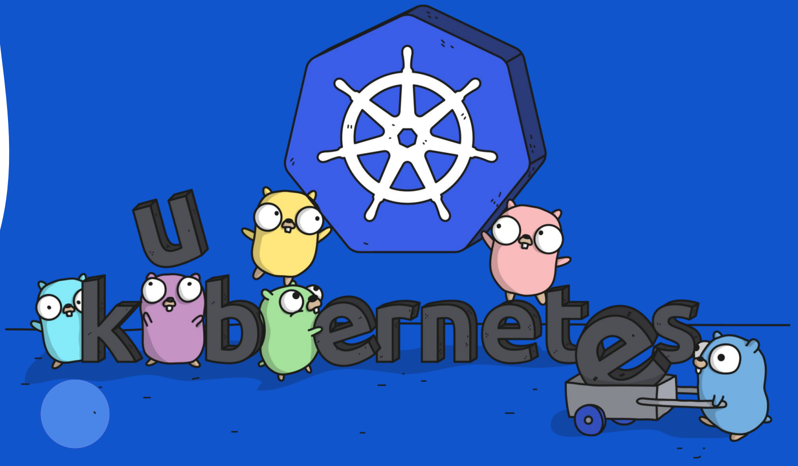
Microservices deployment with Kubernetes



Containers?

Goals?

Kubernetes?

And you?

Whoami

Capgemini

Decathlon

Cloud Native Partners

Agenda

Introduction

Introduction to microservices, and reminder on containerisation.

Kubernetes

Basics on kubernetes, its architecture and these different resources.

Hands-on

Application of the concepts with different labs.

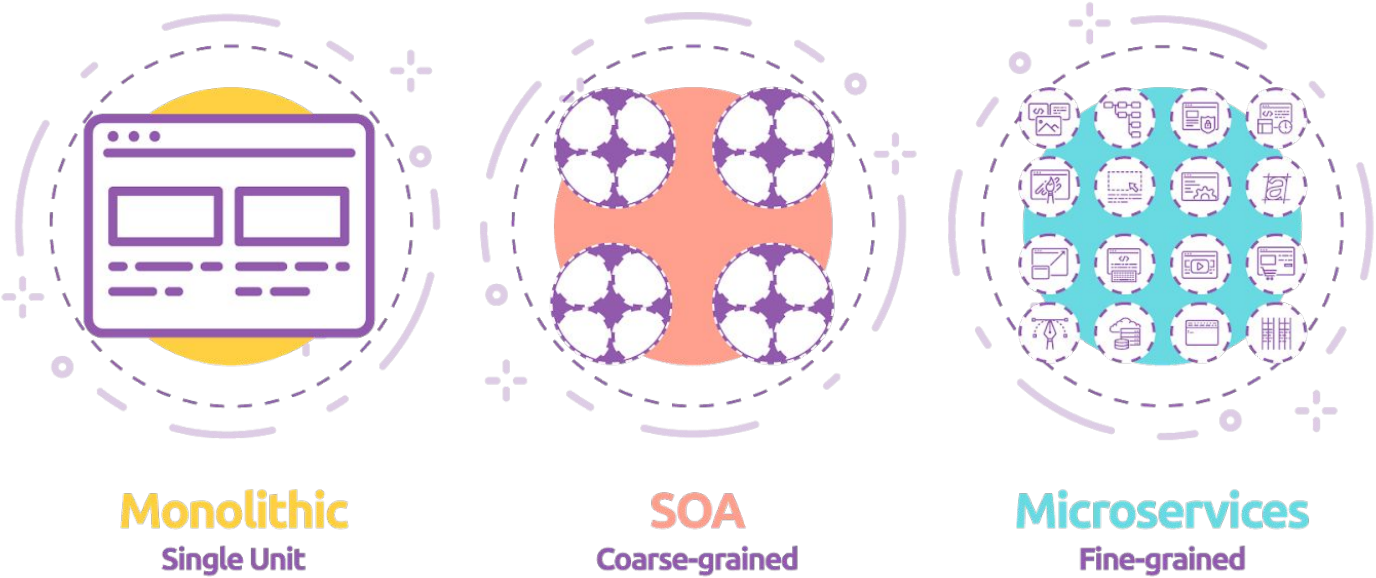
Introduction

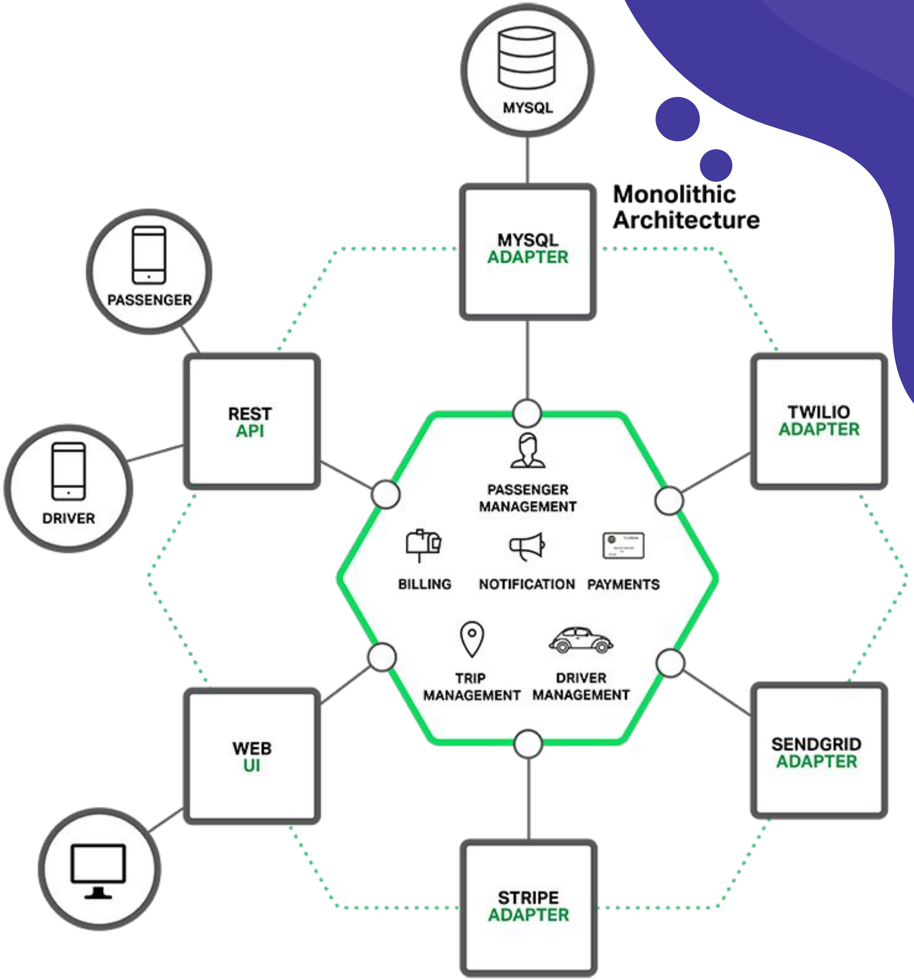
Design Microservices Architecture.

"The golden rule: can you make a change to a service and deploy it by itself without changing anything else?"

-Sam Newman

Monolith VS Microservices





Monolithic application

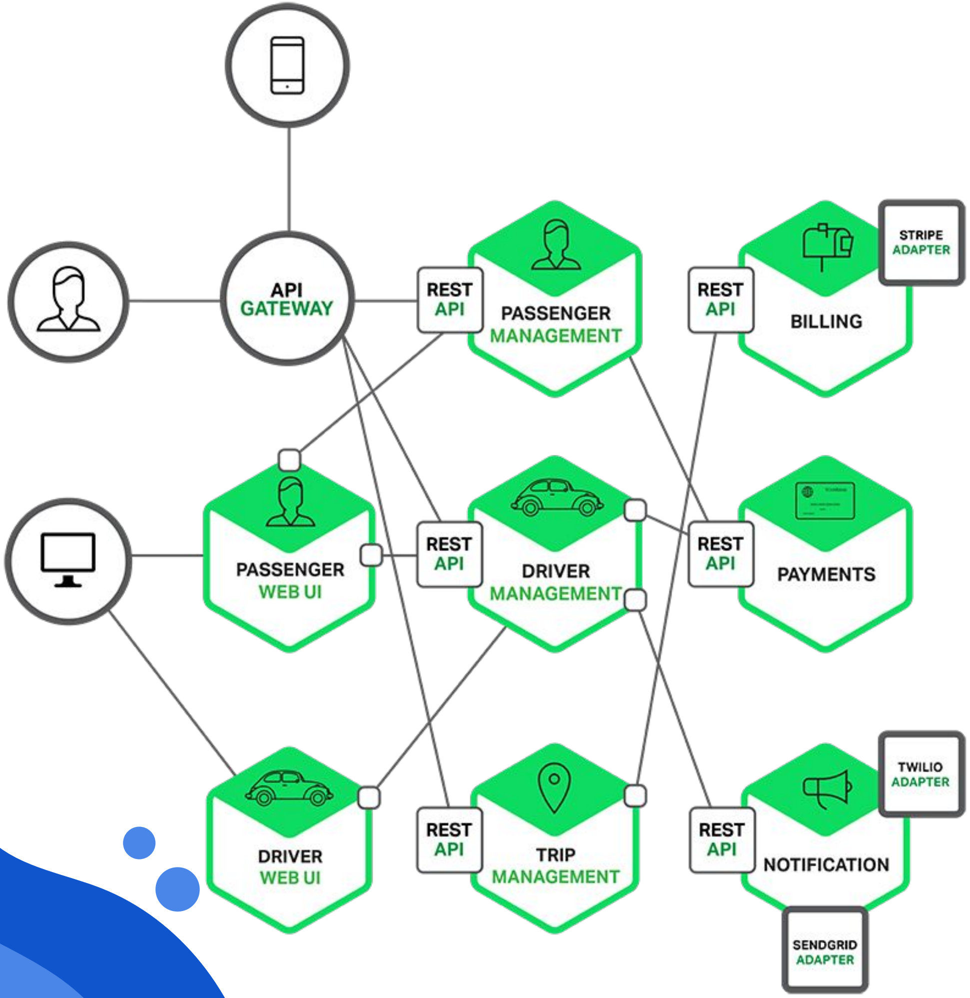
All the software components of an application are assembled together and tightly packaged.

Challenges with monolithic software

Inflexible Monolithic applications cannot be built using different technologies.	**Unreliable** If even one feature of the system does not work, then the entire system does not work.	**Unscalable** Applications cannot be scaled easily since each time the application needs to be updated.
Blocks CD Many features of an application cannot be built and deployed at the same time.	**Slow Development** Development in monolithic applications takes a lot of time to be built.	**Complexity** Features of complex applications have tightly coupled dependencies.

Microservice application

Each service is self-contained and implements a single business capability.



Features of Microservices.

Decoupling Services within a system are largely decoupled, so the application can be easily built.	**Autonomy** Developers and teams can work independently of each other, thus increasing speed.	**Continuous Delivery** Allows frequent releases of software through systematic automation of software creation.
Responsibility Microservices do not focus on applications as projects.	**Decentralized Governance** The focus is on using the right tool for the right job.	**Complexity** Microservices support agile development. Any new feature can be quickly developed.

The good, the bad and the...

Pros

- Ability to scale independently
- Fault tolerance
- Can be swapped out or easily re-written
- Framework and language agnostic
- Adheres to KISS principle
- 12 factors compatible

Cons

- More complexity
- Microservice-based architecture may not provide any meaningful benefits
- No greenfield options.
- more robust methods of testing from the entire engineering team
- The need for increased team management and communication.

Summary

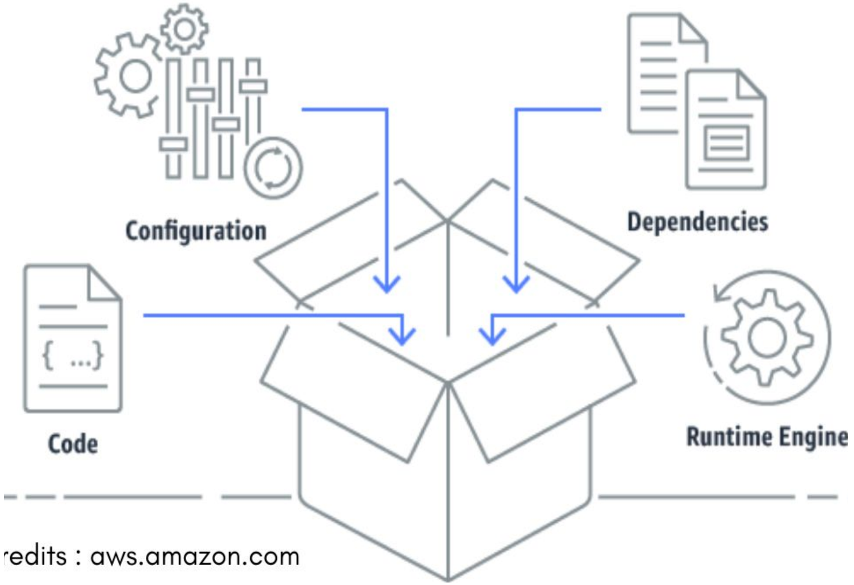
	Pros	Cons
Microservices	Better Organization Decoupled Performance Fewer Mistakes	Cross-cutting Concerns Across Each Service Higher Operational Overhead
Monolith	Fewer Cross-cutting Concerns Less Operational Overhead Performance	Tightly Coupled Harder To Understand

Introduction

Building a microservice in a container

“A container is a software package that contains everything the software needs to run. This includes the executable program as well as system tools, libraries, and settings.”

-Tech Terms

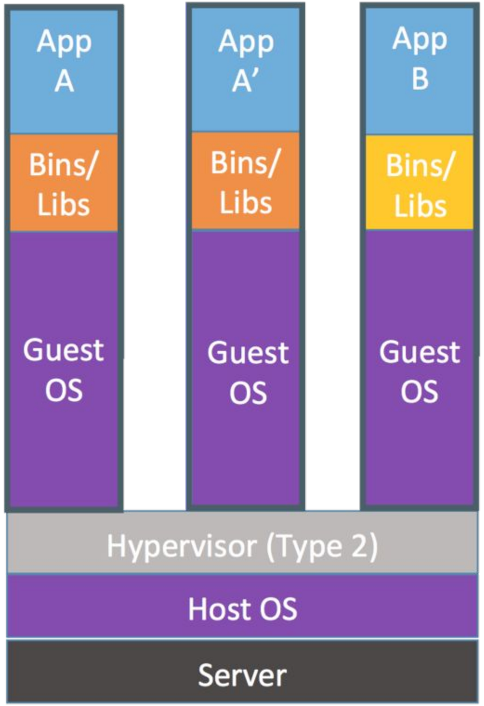


redits : aws.amazon.com

What is a container?

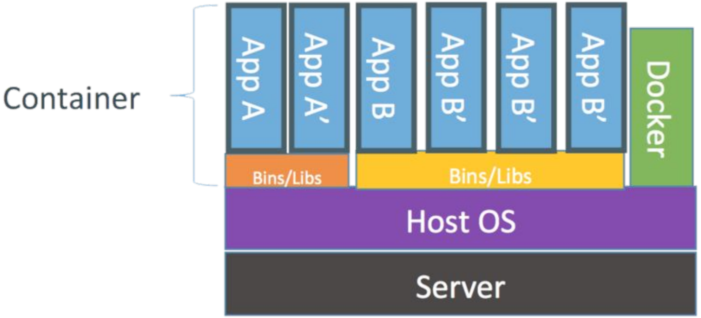
Lightweight, scale-able and isolated VMs in which you run your applications.

Containers VS Virtual machines



Containers are isolated, but share OS and, where appropriate, bins/libraries

...result is significantly faster deployment, much less overhead, easier migration, faster restart



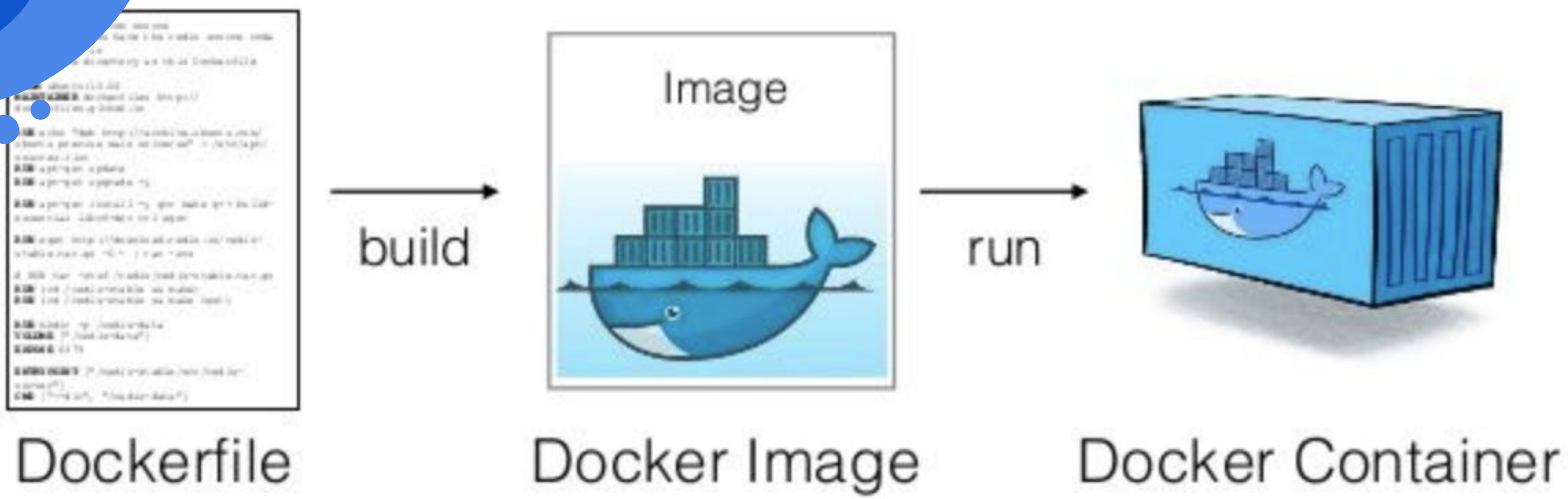
Why it's important?

Devs

- Build once... (finally) run anywhere
- A clean, safe, hygienic, portable runtime environment for your app.
- No worries about missing dependencies, packages during deployments.
- Automate testing, integration, packaging anything you can script.
- Reduce/eliminate concerns about compatibility on different platforms
- A VM without the overhead of a VM. Instant replay and reset of image snapshots.

Ops

- Configure once... run anything
- Make the entire lifecycle more efficient, consistent, and repeatable
- Eliminate inconsistencies between environments.
- Significantly improves the speed and reliability of continuous deployment and continuous integration systems.
- Better performance, costs, deployment, and portability compare to VMs.



WRITE

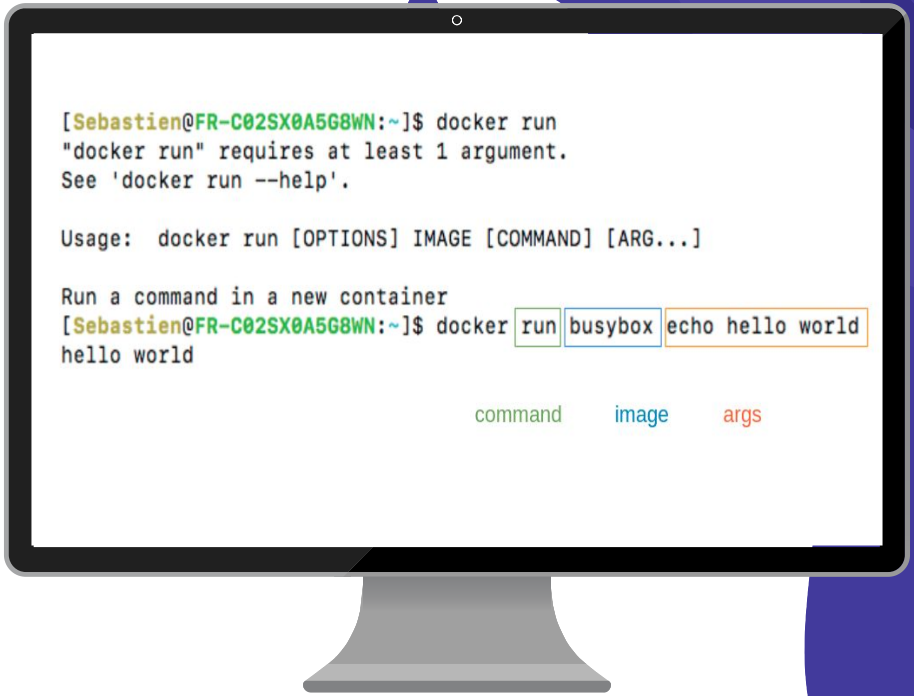
Define the base image, and describe the different tasks to execute.

BUILD

Create your image from your Dockerfile, each step of the script will be executed in order to build the corresponding image.

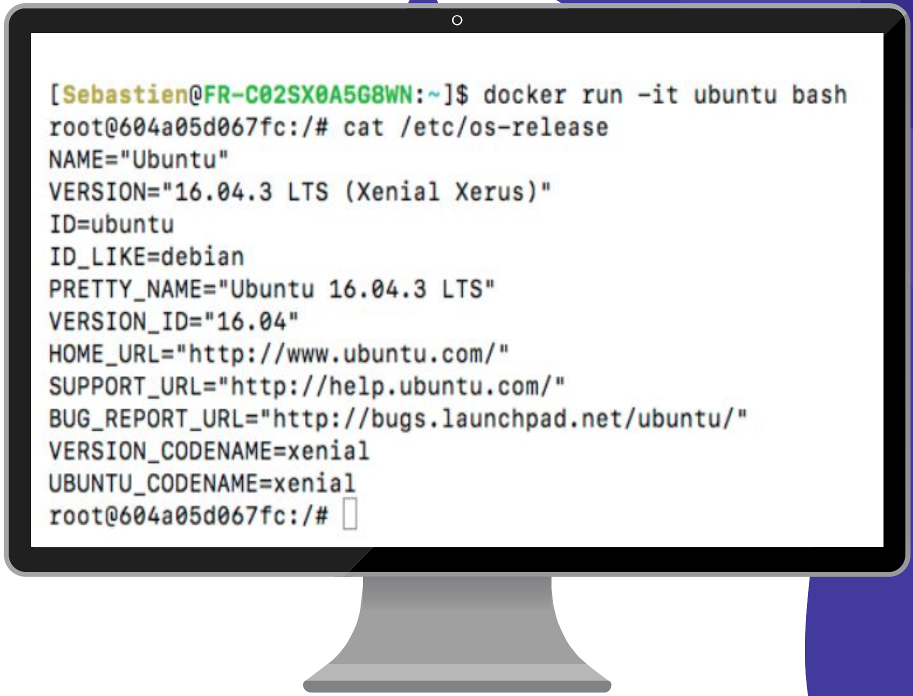
RUN

Start your image on a host, it will execute the main process of the container.



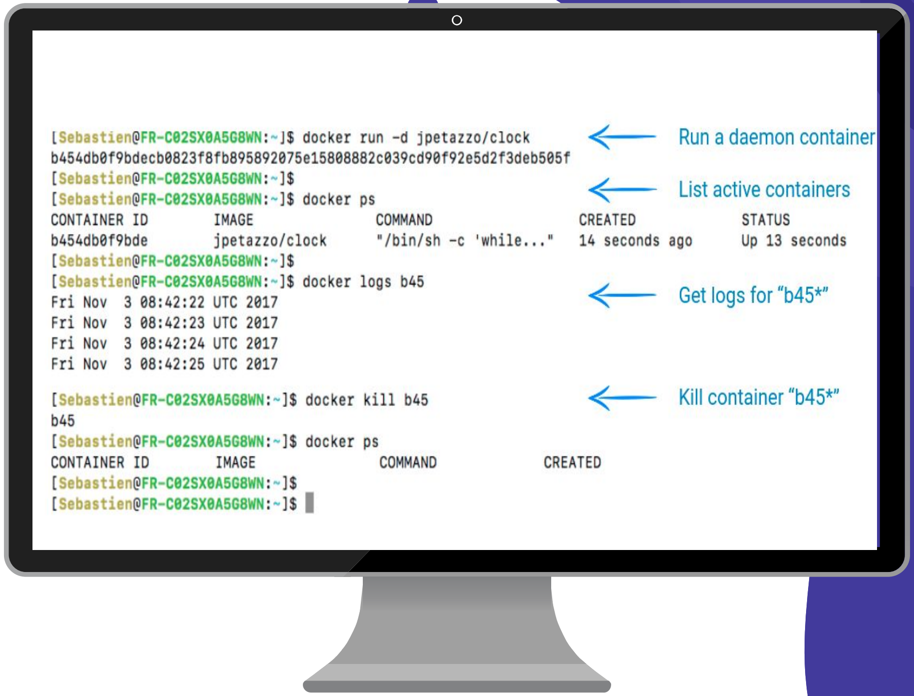
Hello World

let's start our first container!



Let's go inside!

Do not do that in production



Let's daemonize!

that's the point of a container.

https://github.com/skhedim/epsi-k8s/tree/master/docker

Kubernetes

Container Orchestration tool

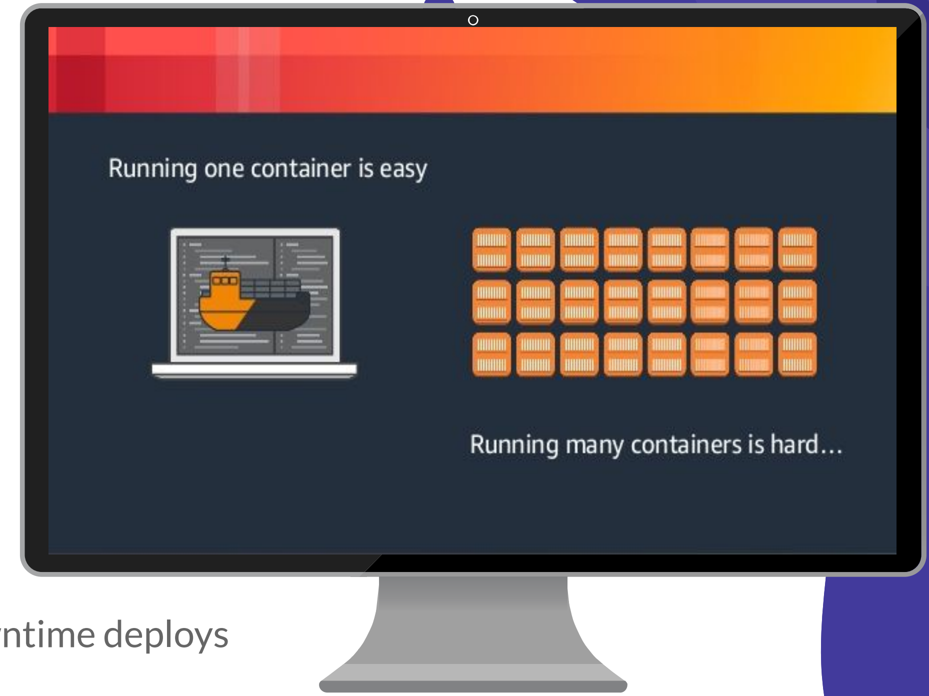
"Kubernetes is the Linux of the cloud"

-Kelsey Hightower

Load Balancing

Scaling

Storage



Zero-downtime deploys

Secrets/configuration

High availability

We need to orchestrate

Container orchestration is all about managing the lifecycles of containers, especially in large, dynamic environments.

With an orchestrator you will be able to:

LIFECYCLE
Choose the right place to run containers.

FAILOVER
Takes care of restarting the components that were stopped abnormally

NETWORK
Manage the IPs of the different pods as well as load balancing.

MUCH MORE
Secrets management
Managing access rights
Configuration files

Many tools, but...

RANCHER

docker SWARM

Nomad

MESOS

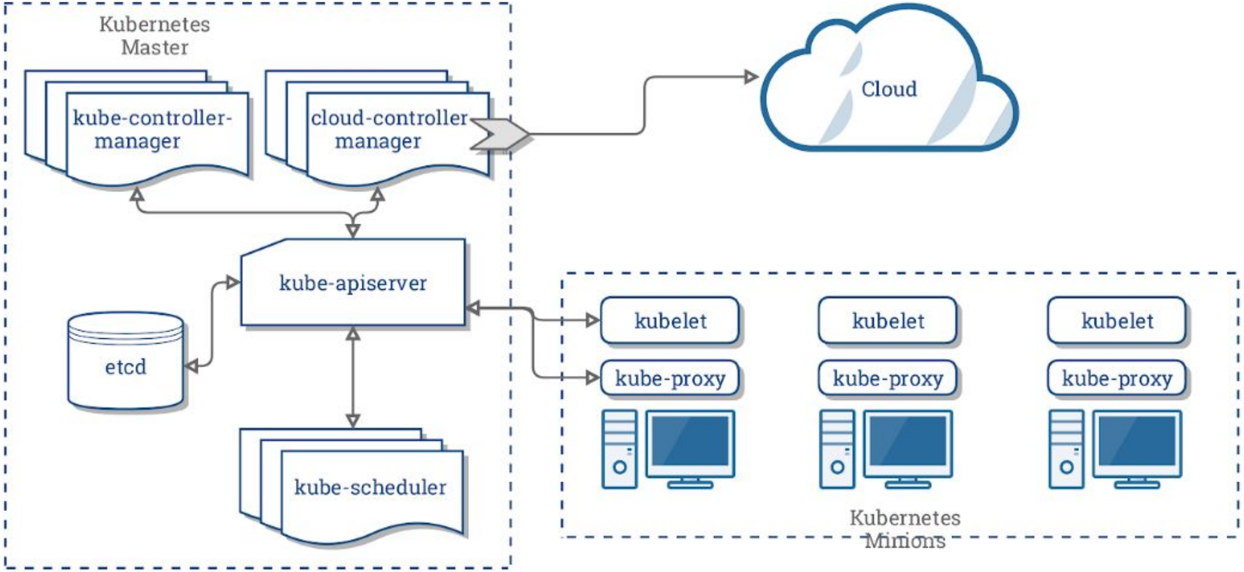
kubernetes

Kubernetes

Architecture

Kubernetes Architecture

Kubernetes itself follows a client-server architecture,

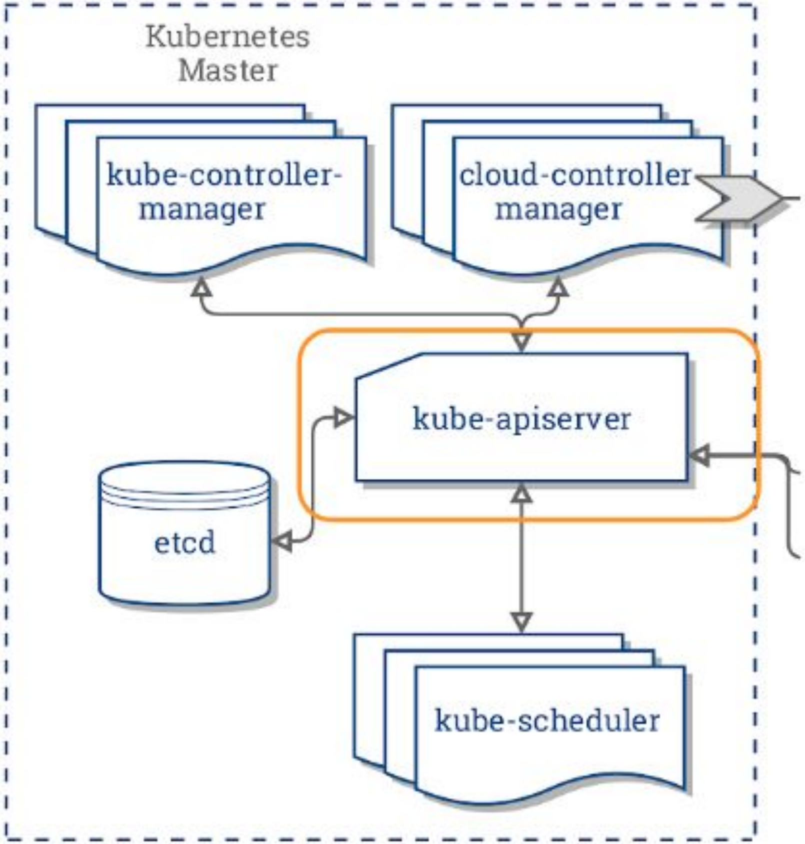


Master:

- ETCD
- kube-apiserver
- kube-controller
- Kube-scheduler
- cloud-controller

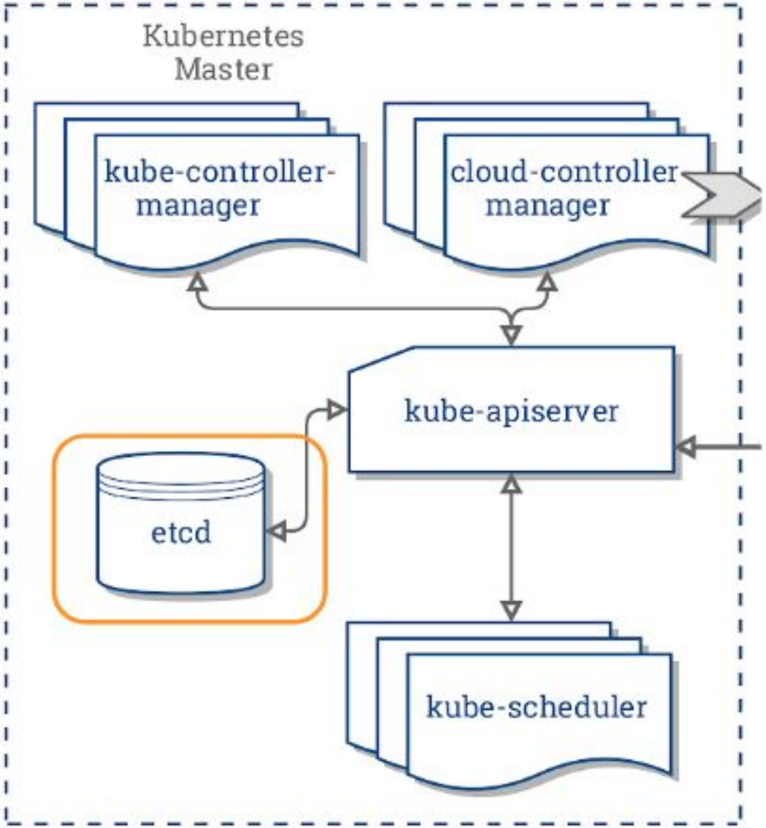
Node:

- kubelet
- kube-proxy



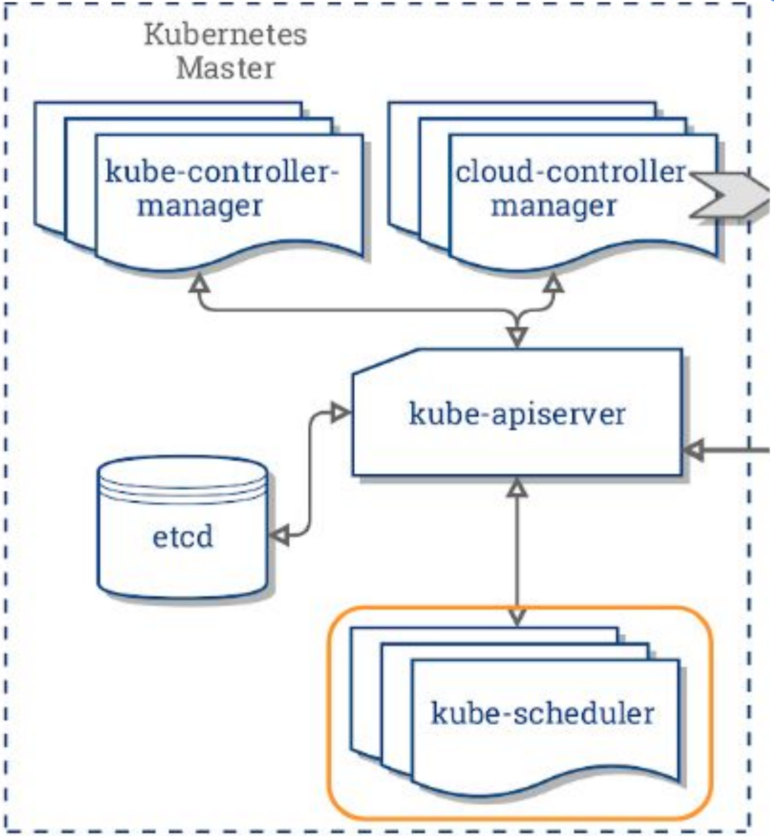
kube-apiserver

Component on the master that exposes the Kubernetes API. It is the front-end for the Kubernetes control plane.



ETCD: Distributed reliable key-value store

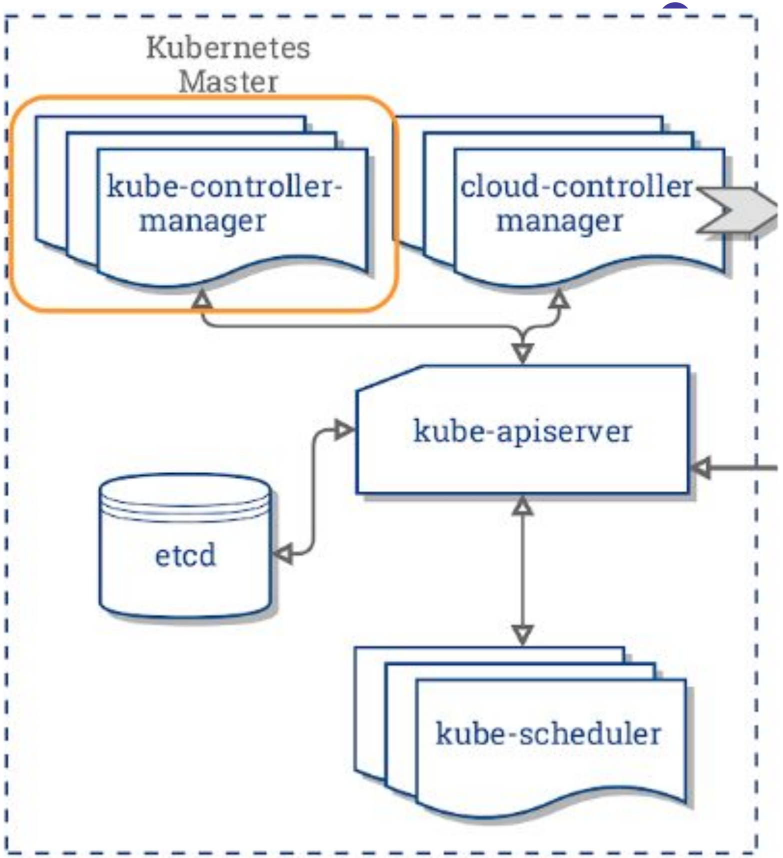
Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.



kube-scheduler

Component on the master that watches newly created pods that have no node assigned, and selects a node for them to run on.

kube-controllers



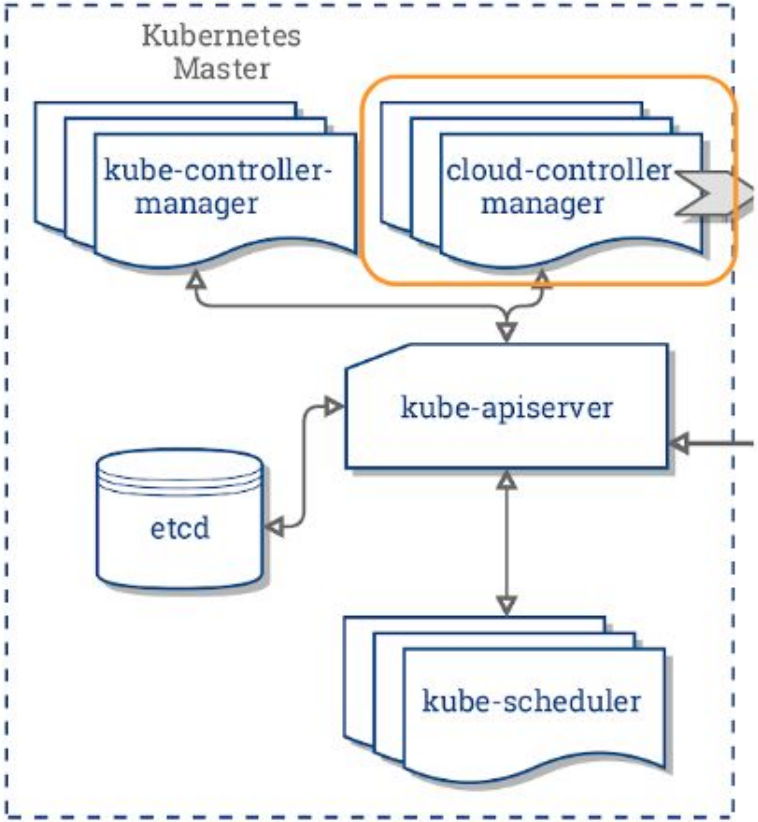
Node Controller: Responsible for noticing and responding when nodes go down.

Replication Controller: Responsible for maintaining the correct number of pods for every replication controller object in the system.

Endpoints Controller: Populates the Endpoints object (that is, joins Services & Pods).

Service Account & Token Controllers: Create default accounts and API access tokens for new namespaces.

cloud-controllers

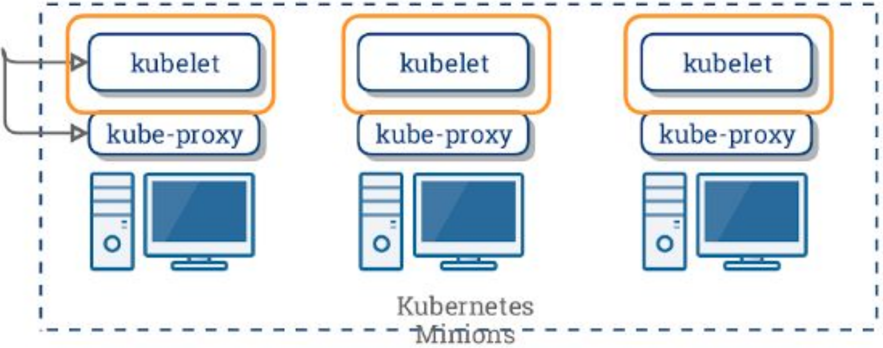


Node Controller: For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding

Route Controller: For setting up routes in the underlying cloud infrastructure

Service Controller: For creating, updating and deleting cloud provider load balancers

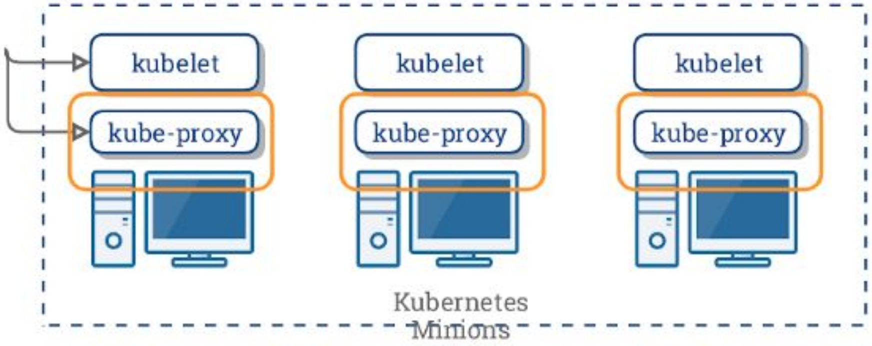
Volume Controller: For creating, attaching, and mounting volumes, and interacting with the cloud provider to orchestrate volumes



kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a pod.

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.



kube-proxy

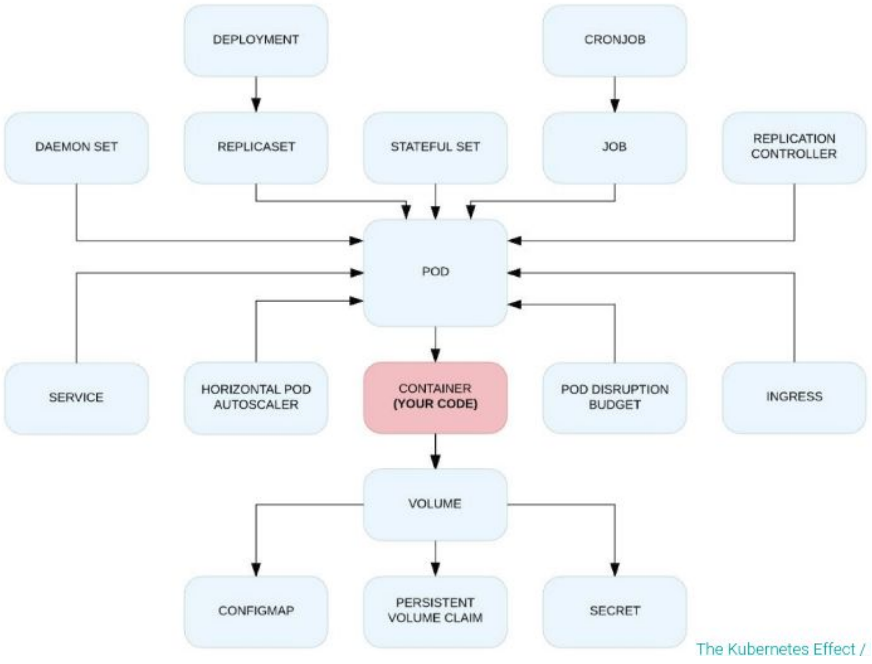
kube-proxy enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding.

Kubernetes

Main features and resources

Kubernetes (commonly stylized as K8s) is an open-source container-orchestration system for automating deployment, scaling and management of containerized applications.

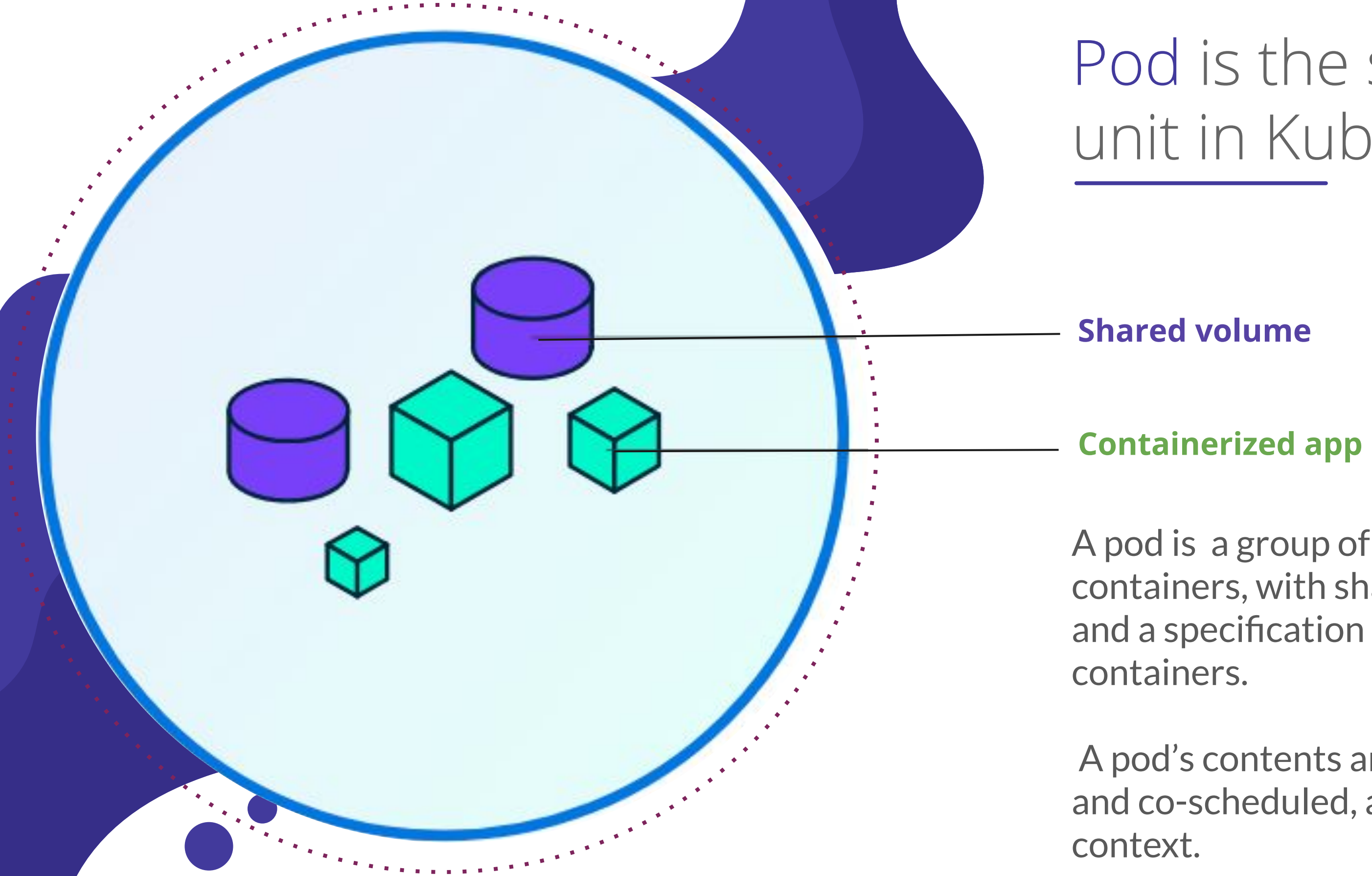
- Wikipedia



Kubernetes objects

Kubernetes objects are persistent entities in the Kubernetes system. Kubernetes uses these entities to represent the state of your cluster.

Pod is the smallest unit in Kubernetes.



A pod is a group of one or more containers, with shared storage/network, and a specification for how to run the containers.

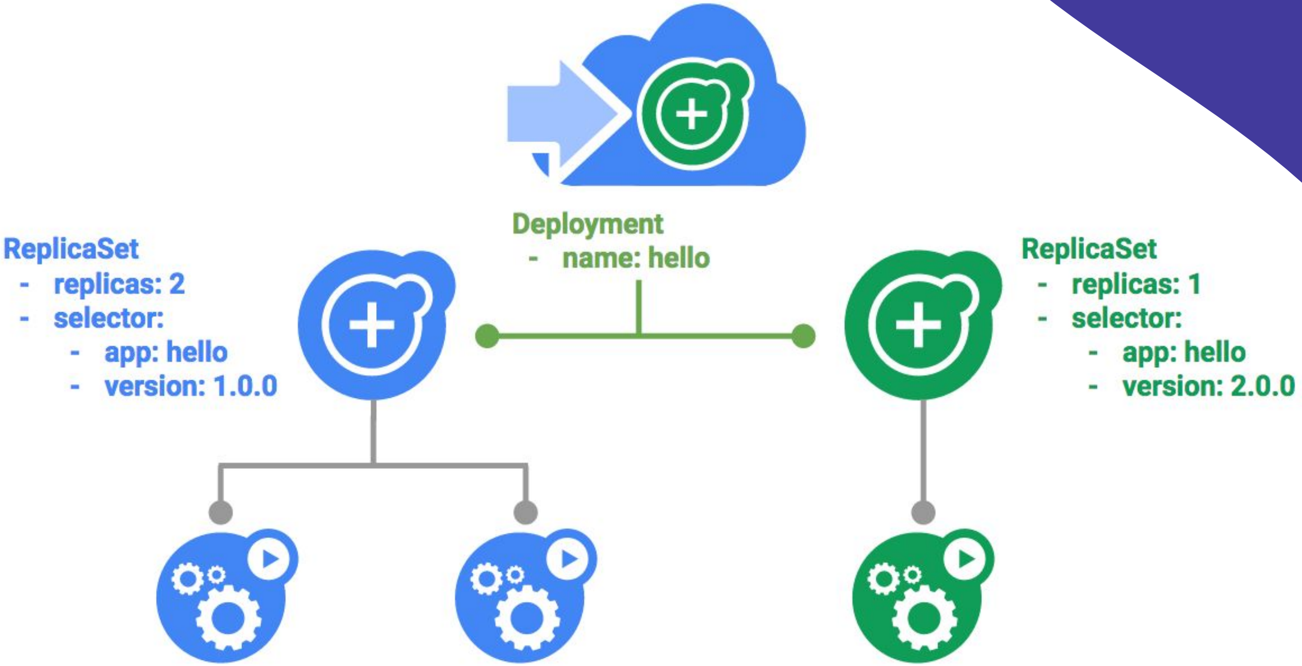
A pod's contents are always co-located and co-scheduled, and run in a shared context.

YAML File

I hope you like the YAML.

```yaml
apiVersion: v1
kind: Pod
metadata:
name: nginx-pod1
labels:
  name: nginx-pod1
spec:
containers:
- name: nginx-pod1
  image: nginx:1.7.1
  ports:
  - name: web
    containerPort: 80
```

ReplicaSet is used to maintain a stable set of replica Pods



Used to guarantee the availability of a specified number of identical Pods.

It's recommended to use Deployments instead of directly using ReplicaSets

YAML File

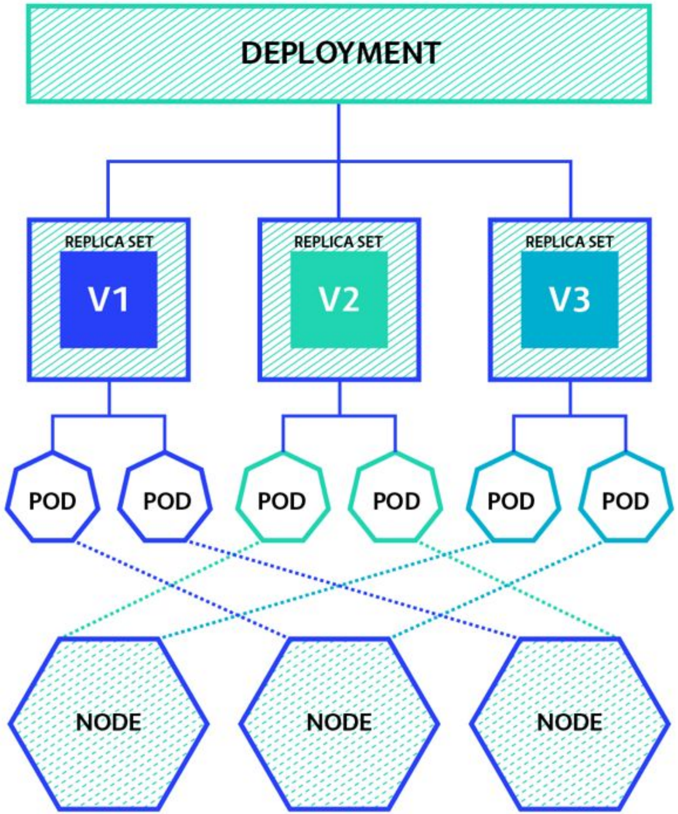
Yeah, it's YAML again.

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
 name: frontend
 labels:
   app: guestbook
spec:
 # modify nb of replicas
 replicas: 3
      spec:
     containers:
     - name: nginx
       image : nginx:1.7.1
```

eployment

vides declarative
dates for Pods and
olicaSets.

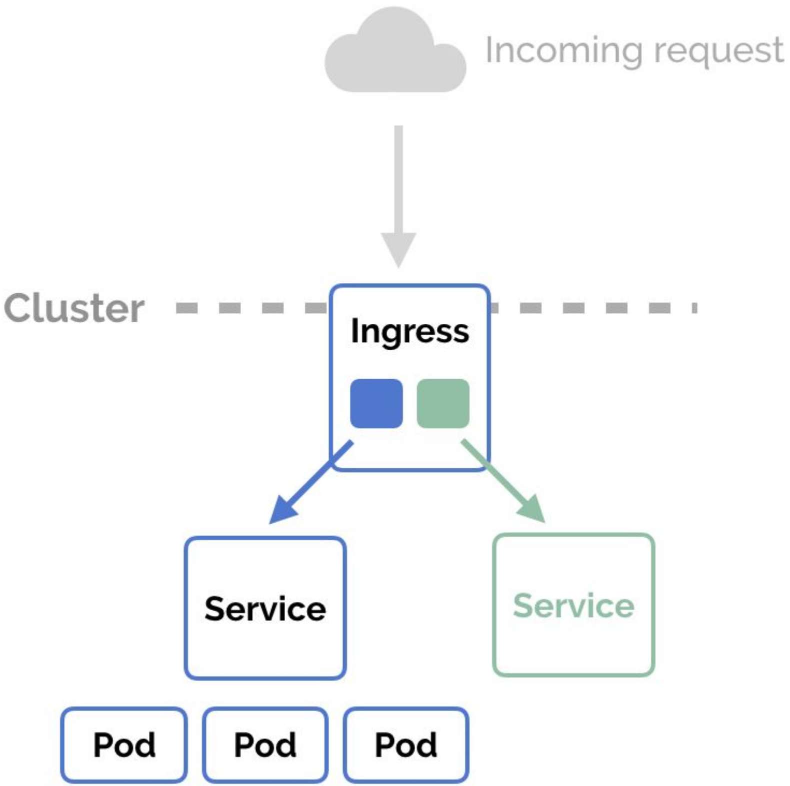
service is self-contained and
ments a single business capability.



YAML File

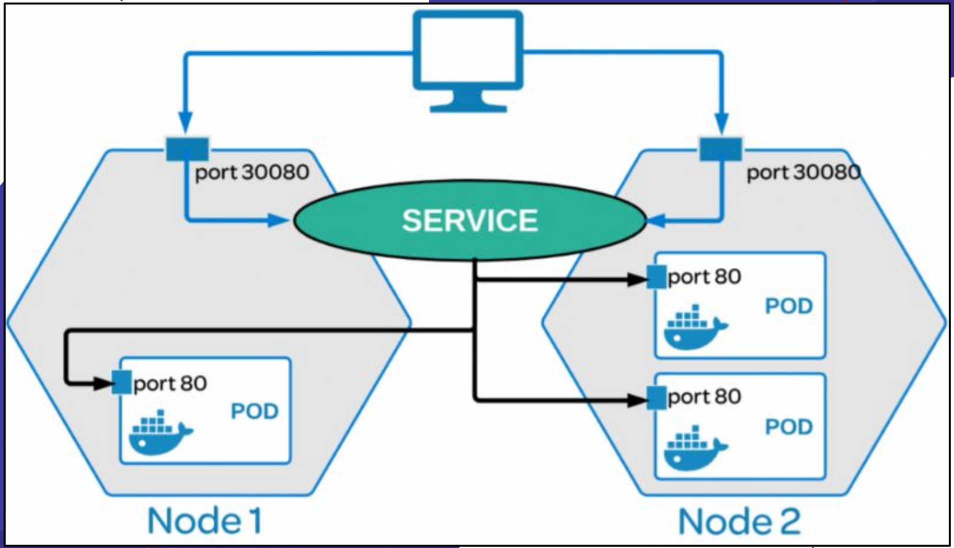
that tablet's getting bigger and bigger, isn't it?

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
 name: nginx-deployment
 labels:
   app: nginx
spec:
 replicas: 3
 selector:
   matchLabels:
     app: nginx
 template:
   metadata:
     labels:
       app: nginx
   spec:
     containers:
     - name: nginx
       image: nginx:1.7.9
       ports:
       - containerPort: 80
```



Network access

An Ingress can be configured to give Services externally-reachable URLs, load balance traffic, terminate SSL / TLS, and offer name based virtual hosting



Persistent Endpoint for Pods.

- A Kubernetes Service is an abstraction which defines a logical set of Pods and a policy by which to access them - sometimes called a micro-service.

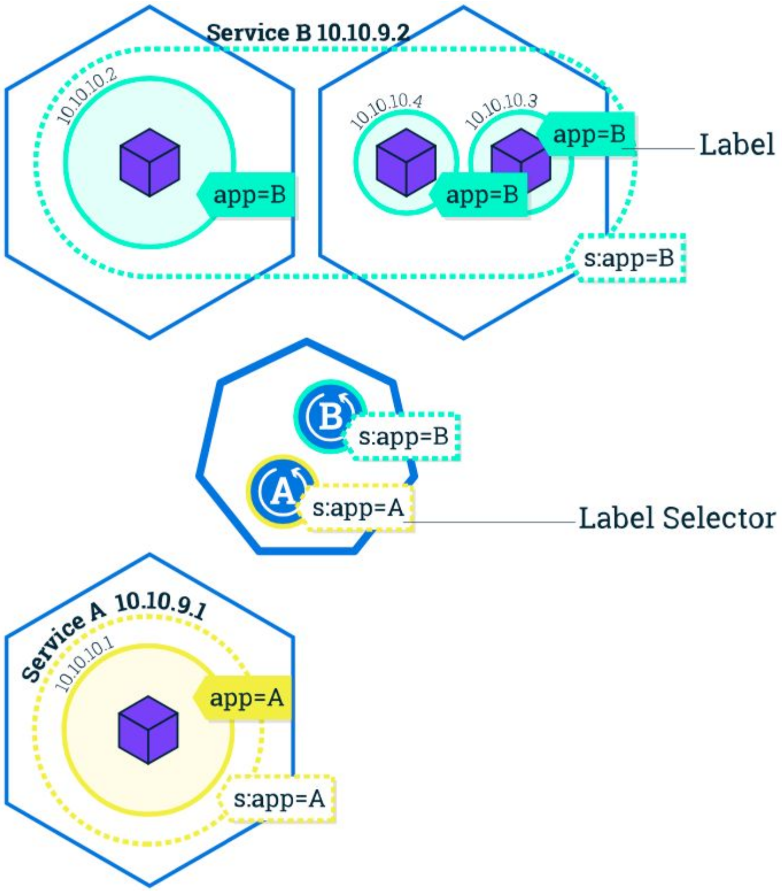
- Use labels to select Pods

blishing Services
rviceTypes)

IP: Exposes the Service on a cluster-internal IP.
ng this value makes the Service only reachable
thin the cluster.

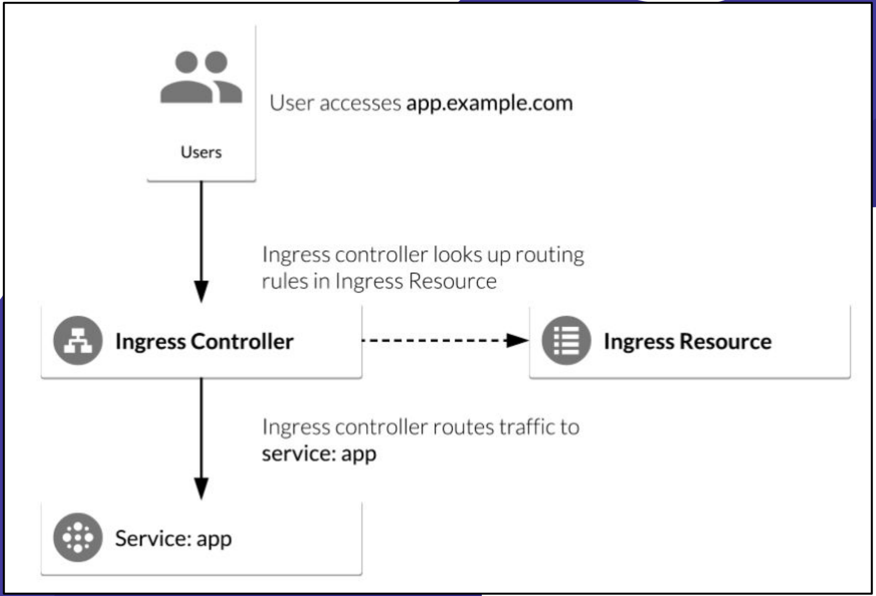
rt: Exposes the Service on each Node's IP at a
ort (the NodePort).

lancer: Exposes the Service externally using a
rovider's load balancer.



YAML File

```yaml
apiVersion: v1
kind: Service
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
    - protocol: TCP
      port: 80
      targetPort: 9376
  clusterIP: 10.0.171.239
  type: LoadBalancer
```



Ingress

An API object that manages external access to the services in a cluster, typically HTTP.

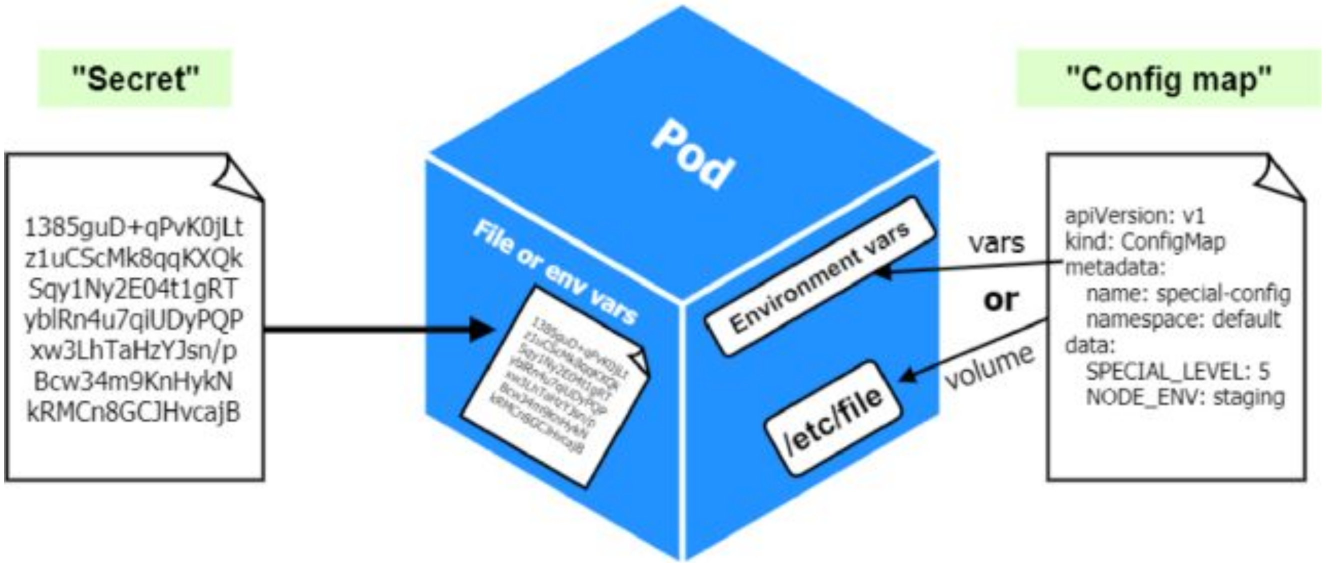
Ingress can provide load balancing, SSL termination and name-based virtual hosting

YAML File

```yaml
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
 name: test-ingress
spec:
 rules:
 - http:
     paths:
     - path: /testpath
       backend:
         serviceName: service
         servicePort: 80
```

Secret and Configmaps

Handling Sensitive Information and Container Configurations



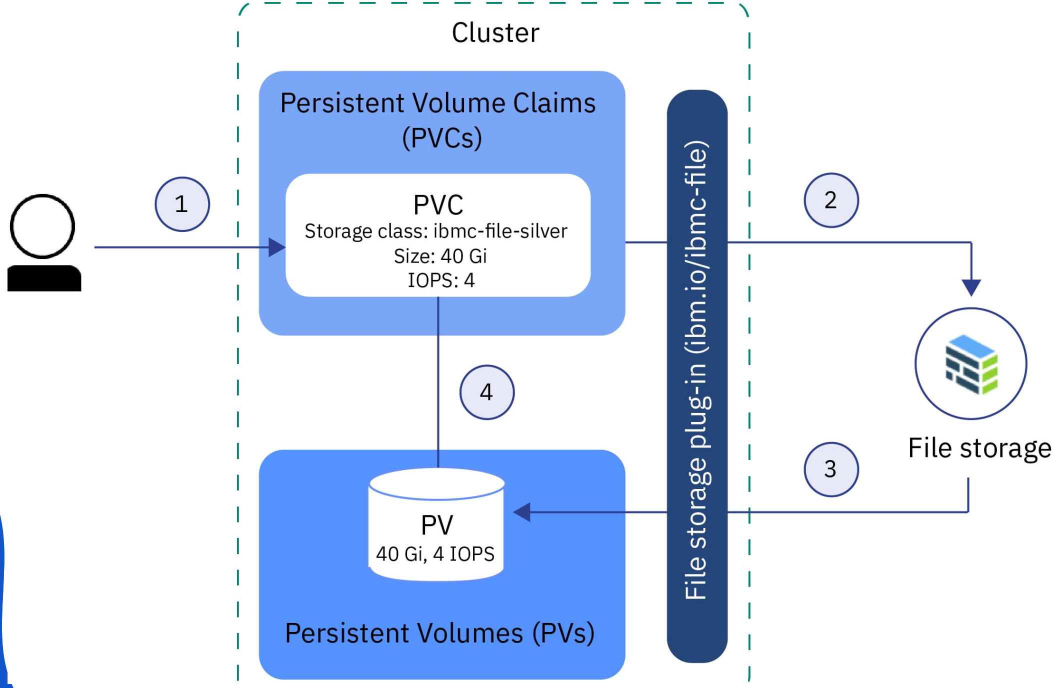
Config Maps: a set of values that can be mapped to a pod as “volume” or passed as environment variables.

Secrets: similar to config maps, secrets can be mounted into a pod as a volume to expose needed information or can be injected as environment variables.

Persist data

ntVolume (PV): resources in the cluster. A storage in the cluster that has been ned by an administrator or dynamically ned using Storage Classes.

ntVolumeClaim (PVC): Request for storage by t is similar to a pod. Pods consume node es and PVCs consume PV resources.



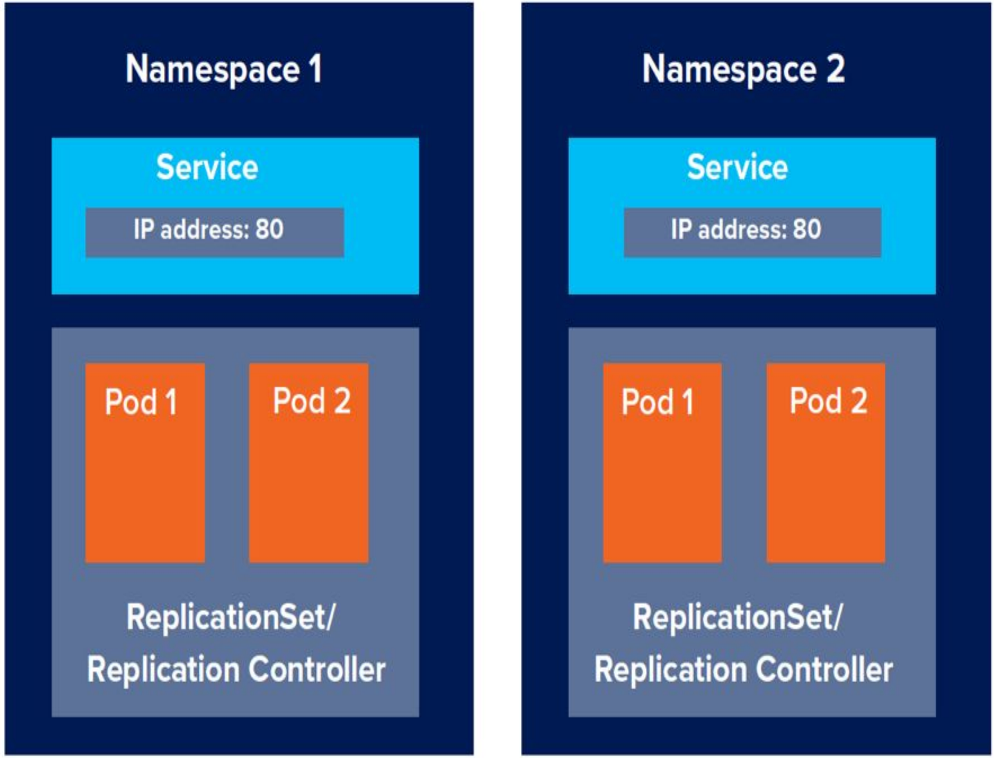
PV

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
name: task-pv-volume
labels:
  type: local
spec:
storageClassName: manual
capacity:
  storage: 10Gi
accessModes:
  - ReadWriteOnce
hostPath:
  path: "/mnt/data"
```

PVC

```yaml
apiVersion: v1
kind:
PersistentVolumeClaim
metadata:
name: myclaim
spec:
accessModes:
  - ReadWriteOnce
volumeMode: Filesystem
resources:
  requests:
    storage: 8Gi
storageClassName:
my-storage-class
```

Namespaces are a logical isolation of the different resources seen previously



Use-case #1: Roles and Responsibilities in an Enterprise

Use-case #2: Using Namespaces to partition development landscapes

Use-case #3: Partitioning of your Customers

So many more!

ClusterRole

ServiceAccounts

mutatingwebhookconfigurations

StorageClass

CronJobs

DaemonSets

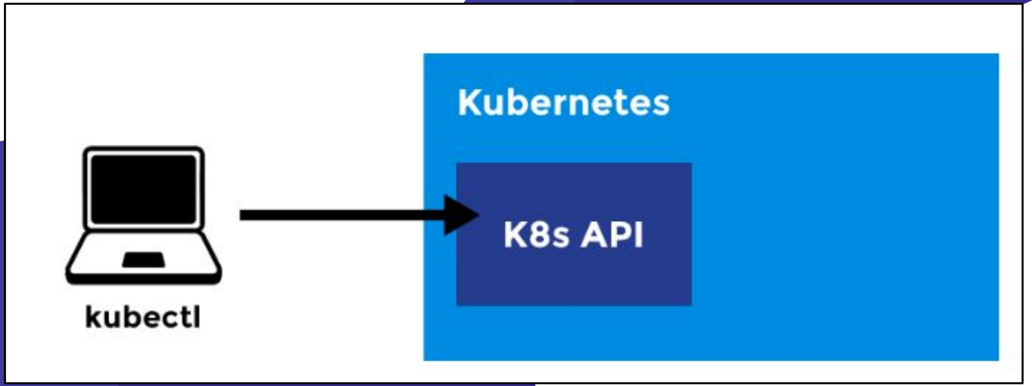
PodDisruptionBudget

ClusterRoleBinding

CRDs

Network policies

Statefulsets



But wait! How can I deploy all of these objects?

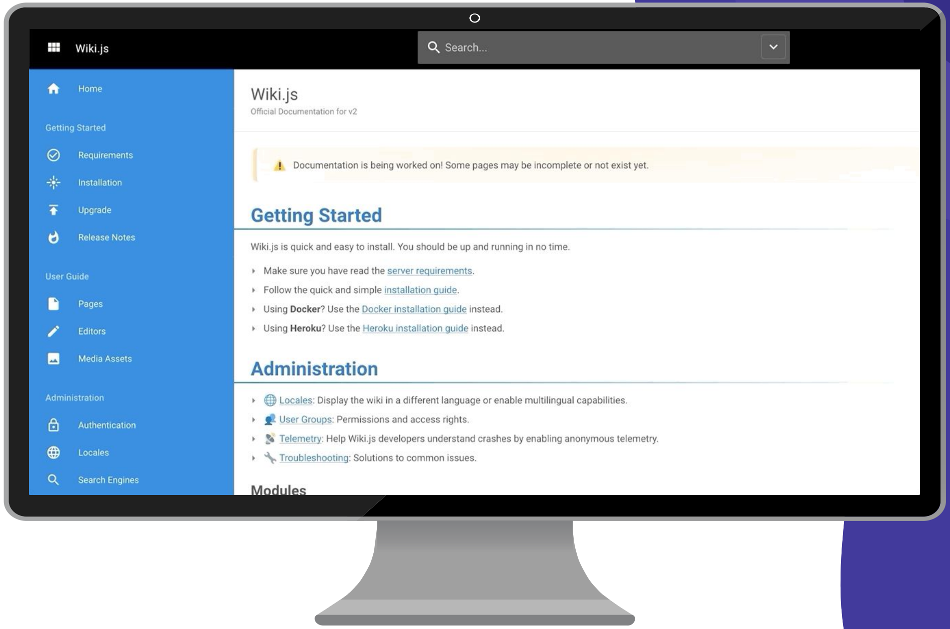
kubectl, a CLI tool to manage your cluster.

- works with a config file
- talks with the API server
- only one tool to do everything

https://github.com/skhedim/epsi-k8s/tree/master/kubernetes

Hands-on

Package an application

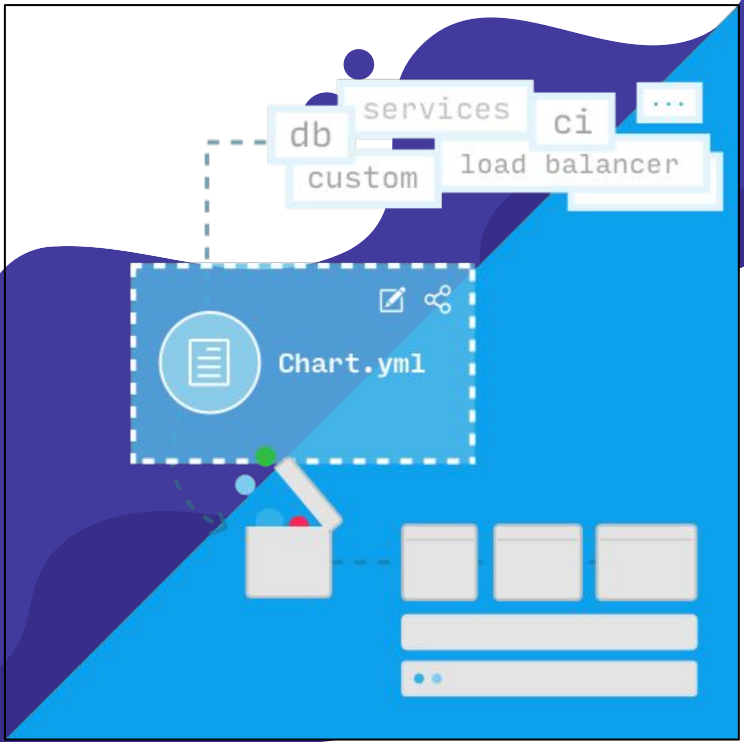


Wiki.js

A wiki engine running on Node.js and written in JavaScript.

There is a docker image of the application

But nothing is packaged for Kubernetes



Package your objects

Helm packages multiple Kubernetes resources into a single object

logical deployment called chart

A Chart is package, a set of k8s resources (deployments, configmaps, services, ingress,....)

A Template is a Kubernetes resource model based on the Golang template engine and the Sprig library

Think about it!

STEP 1

Write your YAML files without variables and test it

STEP 2

Think about the fields that it would be useful to variabilize.

STEP 3

Think of the fields that would be useful to model

STEP 4

Lint each file created, then generate a template https://github.com/skhedim/epsi-k8s/tree/master/helm

Thanks!

Does anyone have any questions?