**Architecture (primitives)**

* What is it?
  + Container Orchestrator
    - Starts and stops container-based apps based on a preconfigured set of rules
    - Higher level of abstraction (‘desired state’ – basically we describe the desired state of our system and the orchestrator figures out how to achieve it)
      * This is done by utilizing controllers (control loops – basically daemons)
      * API Server – the central communication node? (RESTful API over HTTP using JSON)
        + API objects are primitives that represent the system state

**Pods** – one or more containers deployed as a single unit. Basically an app or service.

Can check the health of a pod with a liveness probe.

**Controllers** – make checks and monitor if the system is in the desired state. Create and manage pods.

ReplicaSet – a number of replicas of a particular pod we want up and running

Deployment controller – uhh.. manages rollout of Replica Sets… somehow…

**Services** – provide persistent access point to the applications we deploy in pods. Necessary because of the dynamic nature of Kubernetes (i.e. if one pod goes down and Kubernetes deploys a new one, we need a reliable, seamless way to connect to the pod)

Networking abstraction for pod access and a way to tie a few pods together. So the Service is the front end of the pod(s)? Receives an IP and a DNS name

Basically, usable as pods aggregation and management tools – can act as a load-balancer, point of contact, can scale/remove pods

**Storage** –

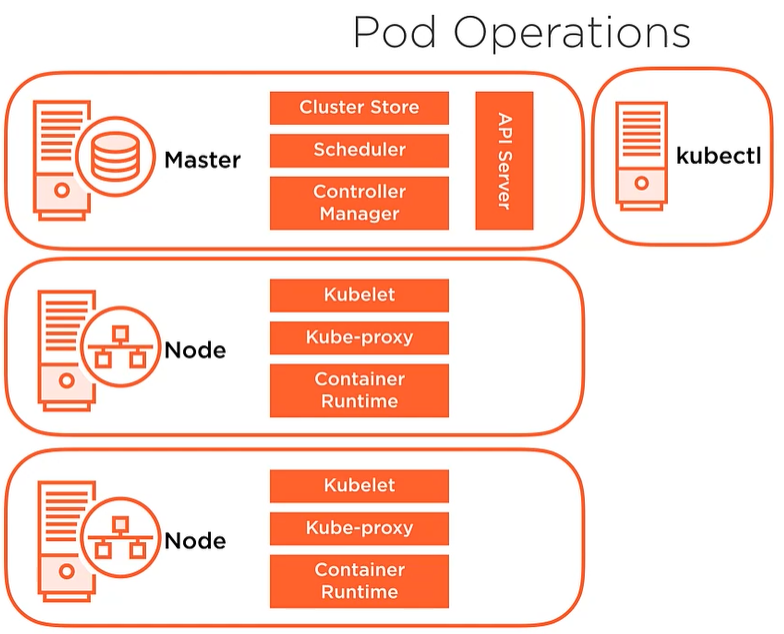
Persistent volume – a dedicated amount of physical storage for a cluster. To make use of that, a pod lays a claim on a portion of that storage.

Nodes - servers that do work

* + - * + They allow us to configure the state of the system, either

Declaratively – describe how we want the system to look (set the desired state)

Imperatively – describe how to get there

* Cluster Components
  + Master (node) - Implements the major control functions of a cluster. Coordinates cluster operations, monitoring and scheduling, and is the primary access point for cluster administration. Comprised of:
    - API Server – the brain of the cluster. Simple RESTful API utilizing basic get, put, post and delete operations.
    - Cluster store (storage) – etcd - the operations we pass to the API server get persisted. This is done by persisting Kubernetes objects into a key-value store called etcd.
    - Scheduler – tells Kubernetes which nods to start pods on, based on the pods resource requirements and other attributes. It basically keeps track of which pod should be where (for example if a newly deployed pod should be on the same node as another(pod affinity), or should not be so(anti-affinity)).
    - Controller manager – implementing lifecycle functions of the controllers that monitor Kubernetes objects. Basically, the high level work to achieve the desired state is done here.
    - kubectl is the utility we use to interact with the API server to manage a cluster
  + Node – where the application pods run (basically a server?)
    - Kubelet – Monitors the API Server for changes. Starts up pods (or takes them down) upon changes passed to the API Server, or upon pre-existing scheduling. Also, reports the node and pod state – done after executing pod liveness probes.
    - Kube-proxy – pod networking and implementing services (so, responsible for the connections between pods) via iptables(?). Routes requests to the appropriate pods (so, also acts as a load balancer).
    - Container Runtime – runtime env for the container image. Pulls the container images from the registry and provides an execution environment for that image and that pod. Whereas the kubelet launches pods, the container runtime launches containers. It is usually Docker (better keep it that way).
  + Scheduled/AddOn components – pods that provide services to the cluster (e.g. DNS pods)
    - DNS – Special Purpose pods provide DNS services inside the cluster
    - Ingress Controllers – advanced HTTP load-balancers and content routers
    - Dashboard – web-based administration of the cluster
* Networking Fundamentals
  + Three rules:
    - All pods (on all nodes) should be able to communicate with each other
    - All nodes should be able to communicate with all pods
    - No Network Address Translation (NAT) – (basically nods and pods should use real IP’s)

Basically, how it works:

1. We pass a request through **kubectl** to the **API server** (e.g. ‘Create 3 of those pods and deploy them in this and that manner’)
2. The **API Server** writes that down in the **Cluster Store**
3. The **Controller Manager** sees that the (recently updated) desired state requires three more pods. Thus, it makes a request for them to the **Scheduler**
4. The **Scheduler** tells the **API Server** that those pods need to be scheduled on these nodes (having assessed which are the best fits)
5. The **kubelet** of the node makes its periodic request to the **API Server** and, if that Node was deemed suitable to host (some of) those pods (i.e. if it has enough CPU/Memory and such to satisfy the requirements of the pods), it deploys them
6. The **Controller Manager** keeps monitoring if all is within the desired parameters.
   1. Any new changes initiated by us loop us back to Step 1.
   2. Any new changes caused by external reasons (e.g. node going down) loop us back to Step 3.

A question to check later

**Services** – provide persistent access point to the applications we deploy in pods. Necessary because of the dynamic nature of Kubernetes (i.e. if one pod goes down and Kubernetes deploys a new one, we need a reliable, seamless way to connect to the pod)

* + What if a Service goes down?
  + Similarly, what if a DNS pod goes down?

Current guess – as services are on pods, they are still monitored by the Controller Manager. As long as **that** doesn’t go down, they will be brought back up just like other pods.

**Installation and Configuration**

Installation

* For Dev purposes, more beginner friendly
  + [Docker Desktop](https://hub.docker.com/editions/community/docker-ce-desktop-windows) & [minikube](https://kubernetes.io/docs/setup/learning-environment/minikube/)
  + [kubeadm](https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/install-kubeadm/)
* For learning a lot, install from scratch
  + [Kubernetes the Hard Way](https://github.com/kelseyhightower/kubernetes-the-hard-way)
* Cloud scenarios
  + AWS EKS
  + Google
  + Azure

**Working with Kubernetes Cluster**

Deploy a resource from a yaml file - kubectl apply -f deployment.yaml

kubectl delete serviceaccount kube2iam

kubectl delete daemonset kube2iam

kubectl delete clusterrole kube2iam

kubectl delete clusterrolebinding kube2iam

kubectl delete deployment csgoempire-service-pod

Enter a pod by name - kubectl exec -it csgoempire-service-pod-676b754b54-wsg9b -- /bin/bash

### install the tools for aws cli

apt-get install -y wget zip less \

&& curl "https://awscli.amazonaws.com/awscli-exe-linux-x86\_64.zip" -o "awscliv2.zip" \

&& unzip awscliv2.zip \

&& ./aws/install \

&& wget <https://s3.amazonaws.com/rds-downloads/rds-combined-ca-bundle.pem>

# Create user in sudoers with password 0

RUN adduser appuser --gecos "" --home "/home/user" --disabled-password \

&& adduser appuser sudo \

&& echo "appuser:0" | chpasswd \

&& addgroup appgroup && adduser appuser appgroup \

&& chown appuser:appgroup /home/user

Making a Helm Chart

Directory should look like:

Chart.yaml, Dockerfile, Values.yaml

Folder templates -> deployment.yaml

helm package C:\Users\BozhidarDimitrov\Desktop\AWS\csgo\CSGO\_repo\infrastructure-as-code\service-pod

helm upgrade --install ${chartName} ${chartFile} --namespace ${params.namespace} ${additionalValueFiles} --version ${dockerAndHelmTAG}"

helm upgrade --install service-pod /home/trackback/service-pod-1-a.tgz

rm service-pod-1-a.tgz

helm package /mnt/c/Users/BozhidarDimitrov/Desktop/AWS/csgo/CSGO\_repo/infrastructure-as-code/service-pod

helm upgrade --install service-pod /home/trackback/service-pod-1-a.tgz

rm service-pod-1-a.tgz

helm delete --purge service-pod

helm package /mnt/c/Users/BozhidarDimitrov/Desktop/AWS/csgo/CSGO\_repo/infrastructure-as-code/service-pod

helm install -n service-pod /home/trackback/service-pod-1-a.tgz

**Docker**

Create a dockerfile – plain file with NO file extension

Simple content examples:

FROM ubuntu:18.04

ENV lang en\_US.UTF-8

#Upgrade repos and install packages

RUN apt-get update && apt-get upgrade -y && \

    apt-get install -y \

    apt-utils \

    python3-pip \

    libssl-dev \

    openssl \

    libpq-dev \

    curl \

    telnet \

    mysql-client

CMD echo "Running"

CMD tail -f /dev/null

# Create non-root user

RUN adduser appuser --disabled-login --gecos "" --home "/app" --no-create-home \

    && addgroup appgroup && adduser appuser appgroup

RUN mkdir /app && chown appuser:appgroup /app

USER appuser

# Copy our source code into the container

COPY --chown=appuser:appgroup ./webserver /app

WORKDIR /app

RUN pip3 install -r requirements.txt

FROM - gives us the base the container will run on – the underlying OS

ENV – sets environment variables, in this example text encoding

RUN – set the container conditions – what tools will be needed to install in the image, what directories and permissions

COPY – copies file contents (from where?!? – probably from the location of the dockerfile) to designated location

USER – the user that will be used

WORKDIR – the directory stuff will be executed from (relative path startpoint)

CMD – commands. Tail -f /dev/null is used to keep the container running

To create an image from the dockerfile – docker build -t [image name] [location of dockerfile]

To see all images – docker images -a

To rename an image – docker tag <old\_name> <new\_name>

* docker rmi <old\_name>

To delete an image – docker rmi <image\_id>

To push the image to the ECR repo –

* Authenticate to AWS
* - aws ecr get-login-password --region us-east-2 | docker login --username AWS --password-stdin 366897636402.dkr.ecr.us-east-2.amazonaws.com
* Tag the built image
* docker tag service-pod:latest 366897636402.dkr.ecr.us-east-2.amazonaws.com/service-pod:latest
* Push to ECR
* docker push 366897636402.dkr.ecr.us-east-2.amazonaws.com/service-pod:latest

To run a container from docker image – docker run [image name]