Kubernetes Basics

Scenario: We have a Kubernetes cluster having two Pods within, One pod is my-app (the main application) and the other Pod is DB (a database Pod having DB for my-app). Both communicate between themselves via Service.

Basic Definitions

Node: A node is a virtual machine, and it can have Pods (We can have an application pod which requires another Database Pod for its function).

Pod: An abstraction over an container (has container within). Creates a running environment or a layer on top of a container. Can run multiple application containers (this happens only in case of one main application container and a helper container or a side service container) but usually 1 container per Pod.

Kubernetes offers out of the box virtual network which means that each pod(not the container) gets its own IP address, and each Pod can communicate among themselves using this IP address(internal IP).

Pods Components in Kubernetes are Ephemeral, which means they can die very easily. When a pod dies a new Pod will created in its place and a new IP address is created upon re-creation of the Pod, which makes it inconvenient when communicating between Pods as we have to adjust the IP every time when a Pod restarts. To overcome the difficulty we have another component of Kubernetes called "Service".

Service: It is a static IP address or Permanent IP address that can be attached to each Pod. Lifecycle of Pod and Service are not connected in any way, so if a Pod dies the Service and the IP address will still be the same and does not change. Service also works as a load-balancer between replicas as multiple replicas of same Pod use same Service and this helps to distribute traffic between the Pods.

For the application to be accessible via an external browser we need an external service. An external service is a service that opens the communication from external sources, but we would not want our DB pods to be accessible for external communications, so for that we make use of an Internal service for that we specify when creating one.

While creating an external service we can access the application via an url of numbers(the node ip address followed by port number of the service), which is not good looking so to overcome this we make use of a kubernetes component called "Ingress".

Ingress: Instead of the request to application directly requesting the service, the request goes to the Ingress which forwards the request to the service.

We know Pods communicate with each other using a service so an application will have a database endpoint (for e.g.: mongodb service) which it uses to communicate with the database but where do we configure the database url or endpoint?, Traditionally we configure it within our application, properties file of the application. But what if the endpoint of service or service name has changed(for eg: from mongodb service to mongodb), we would need to rebuild the application with new version and push it up to the repository and we will need to pull that new image into the Pod and restart all over again. This is a tedious process for a very small change, so for that purpose we make use of a Kubernetes component called as "ConfigMap".

ConfigMap: It is basically an external configuration to the application, it usually contains configurations like URLs of DB or some other services that we use. In Kubernetes we just connect ConfigMap with the Pod, so that Pod actually gets the data that ConfigMap contains.

Sometimes part of the external configuration can be something like a DB username and Password which may also change in the process of application deployment, but putting these kinds of sensitive data onto a ConfigMap can make it insecure, for this purpose we can make use of a Kubernetes component called "Secret".

Secret: It is similar to a ConfigMap but the difference is it is used to store secret data (Eg: Credentials), these are not stored in plain text but in base64 encoded format.

Data Storage in Kubernetes

We have a DB Pod that an application uses and it has some data or it generates some data. In this case if the DB container or Pod gets restarted the data will be gone, but we need our data to persist for reliably long term. To achieve this we will make use of another Kubernetes component called "Volumes".

Volumes: It attaches a physical storage on a hard drive to a Pod, the storage could either be on a local machine(meaning on the same server node where the Pod is running) or it can be on a remote storage(meaning outside of Kubernetes cluster).

Assume the application is successfully running and what if a user wants to access the application when the Pod is deleted? This will create a downtime for the application, which is not something we desire. To avoid downtime we can make use of replicas of our Pod. This is possible because multiple replicas of same Pod have same Service which also works as a load balancer and sends oncoming traffic to available Pods/ least busy Pods. In order to create a replica for our Pod we don't need to create a second Pod but

rather we just define a blueprint for the Pod to replicate and specify how many replicas of the Pod we would like to run. The blueprint is called "Deployment" which is another component of Kubernetes.

Deployment: We don't directly work with Pods, but instead we create deployments. We know Pods is a layer of abstraction over containers and Deployments are a layer of abstraction over Pods.

Now that the application is available for the user we can access it with no downtime, but for the application to work well we need DB Pod to work, but we cannot replicate DB using deployments because database has a *state* which is it's data. This means that if we have replicas of the database they would all need to access the same shared data storage and here we need some kind of a mechanism to manage which Pods are writing to the storage currently or which Pods are reading from the storage to avoid data inconsistencies. This mechanism in addition with replicating feature is provided by another Kubernetes component called "Stateful Set".

StatefulSet: It is meant specifically for applications like Databases. Deploying a StatefulSet in kubernetes is not easy; hence it is common to have DB remotely and only stateless apps within kubernetes.

!!!Note!!! : - Deployments for stateless apps, StatefulSet for stateful apps.

KUBERNETES ARCHITECTURE

Scenario: We will start with a node from the previous scenario itself. A node with my-app and DB pods

Worker Machine:

A Node is called as a worker machine in Kubernetes cluster. Each node has multiple Pods in it. Three processes must be installed in every node that are used to schedule and manage the Pods. Worker Nodes do the actual work.

The three processes that need to run on the node are:

1. Container Runtime - Runtime for application Pod having containers running inside it.(e.g.: docker)

2. Kubelet - schedules Pods and containers, has interface to interact with the container runtime and the node itself. Responsible for taking the configuration and running the Pod or starting a Pod with a container inside and assigning resources from th node to the container(Eg: CPU,Storage,RAM..)

3. Kube Proxy (k-proxy) - Responsible for forwarding request from services to Pods.

Master Nodes/ Processes:

A Master Node is different from a worker node. There are 4 processes that run on every master node that controls the cluster state and worker node as well. The 4 processes are:

1. API Server - When a user needs to deploy a new application in a Kubernetes cluster we interact with the API server using some client (E.g.: UI, Kubernetes Dashboard, CLI...). API server works like a cluster gateway

2. Scheduler - Decides to which worker nodes. It looks at the request incoming for Pod creation via the API server, checks the resources required by the application, wit this info it will see the worker nodes and checks them for available resources on each worker node and will schedule the Pod to the node that is least busy.

3. Controller Manager - Detects cluster state changes(Eg: Pods crashing). When a Pod dies the cluster manager detects it and tries to recover the cluster state, for that it makes a request to the scheduler to reschedule the dead pods.

4. Etcd - It is a key value store of a cluster state. It can taken as a cluster brain, which means that every change in the cluster (Eg: When a new pod gets scheduled, when a pod dies all of these changes) gets saved or updated into the key-value store of etcd. But it does not store any of the actual application data.

In practice a Kubernetes cluster is usually made up of multiple masters where each masternode runs its master processes, here the API server is load balanced and the etcd store forms distributed storage across all the master nodes.

Example Cluster Setup

In a very small cluster we will have:

Two Master Nodes
Three Worker Nodes

Note that the hardware resources of Master and Node servers differ, the master processes are more important but they actually have less workload hence less resources is required. Whereas worker nodes do the actual job of running the Pods with containers inside them hence they require more resources. As application complexity and demand of resources increases we may have more master and node servers to the cluster, thus forming a more robust cluster to satisfy the application requirements.

To add new Master/Slave nodes:

Step1: Get a new bare server
Step2: Install all master/worker node processes
Step3: Add it to the cluster

Minikube & kubectl - Local Setup

In a Production Cluster Setup we have Multiple Masters and Workers Nodes. But what if we want to test it on our local machine, setting up the entire cluster as said will be difficult and impossible without proper resources. So for this use case we can use an Open Source tool called "mini-kube".

Minikube:

It is basically a one node cluster, where the master processes and worker processes run on the same node. The node will have a docker container runtime pre-installed. The way it will run on our system is via a virtual-box or some other hypervisor. In simple words, minikube will create a virtual box on our system and the one node runs on that virtual-box. This can be used for testing Kubernetes on our local setup.

With this we can setup a minikube cluster on our system but with our local machine we need a way to interact with the cluster so we will create components, configure tem, etc.. This is where "kubectl" comes into place.

kubectl:

It is a CLI tool for Kubernetes cluster. It interacts with the API server to create, delete and maintain the Pods in the cluster.

Installation and Creation of Minikube cluster [Hands-On]

Minikube Installation Guide: https://minikube.sigs.k8s.io/docs/start/
kubectl Installation Guide: https://kubernetes.io/docs/tasks/tools/
Install a hypervisor anything that suits: Virtual Box recommended

Step1: Install hyperkit (hypervisor)
Step2: Install minikube (installs kubectl as well, because minikube has kubectl as dependency)
Step3: To check installation type "kubectl" and run, the type "minikube" and run.

Starting a minikube cluster:

Step1: minikube start --vm-driver=hyperkit (tells the minikube to use hyperkit to start cluster)
Step2: minikube has docker runtime pre-installed so it does not require specific installation.

Main kubectl commands

Commands:

Create Kubernetes components: kubectl create -h (Provides a list of create commands)
In kubernetes we can't create Pods directly hence we create a deploymnt which in turn creates a pod.

To create a deployment:

Syntax: kubectl create deployment <name> --image=<image> [--dry-run] [options]
Example: kubectl create deployment nginx-depl --image=nginx
This will download the latest nginx image from docker hub
- blueprint for creating pods
- most basic configuration for deployment(name and image to use)
- rest set to defaults
Between the deployment and pod we have a layer called replicaset

To get ReplicaSet: kubectl get replicaset

To get deployment: kubectl get deployment

To get Pod: kubectl get pod (check if the status of the cluster is running else re-run the command)

Layers of Abstraction:

Deployment manages ReplicaSet
ReplicaSet manages a Pod
Pod is an abstraction of Container

Everything below Deployment needs to be handled by Kubernetes

CRUD in Kubernetes

To edit deployment:
Syntax: kubectl edit deployment <cluster_name>
Example: kubectl edit deployment nginx-depl

To get logs of pod:
Syntax: kubectl logs <pod_name>
No output as there are no logs

Create a new deployment for mongodb:
kubectl create deployment mongo-depl --image=mongo

To get Pod description:

kubectl describe <pod_name>

To check what's going inside a pod:

kubectl exec -it <pod_name> --bin/bash (it means interactive terminal).
Returns the terminal of application container of the pod specified.
This is useful in debugging or to test something or try something.

To exits: #exit

To delete pods: kubectl delete deployment <deployment_name>

To set a custom deployment file: kubectl apply -f <file_name>
Create the file, Open the file, paste the basic deployment configuration.
Kubernetes knows when to create a new deployment and when to update existing.

Kubernetes Configuration File

Parts of a K8 configuration file:

1. Metadata - Has metadata of the component we create (name, labels)
2. Specification - Has the configurations that we want to apply. Attributes here are specific to the kind of component.
3. Status (Auto-Generated and added by Kubernetes) - Compares desired state and actual state, this gets data from the etcd.

Format of configuration file is yaml, it is a human friendly data serialization standard for all programming languages. It has strict indentation.

Blueprint for Pods (Template):

It is basically a configuration file inside a configuration file. template has it's own metadata and spec to work with.

Labels and Selectors:

The connection is established using labels and selectors. The metadata has the labels and the spec contains the selectors.

A Label provides a key-value pair for the component (app:nginx) and the label just sticks to the component, so we give pods created with blueprint the label (app:nginx) and tell the deployment to connect (matchLabels) or match all the labels with app:nginx to create the connection. This way a deployment will know the pods connected to it.

Now deployment has its own label(app:nginx) and these two labels are used by the service Selector, so in specification of service a Selector is defined which basically makes a connection between the service and the deployment(or its pods).

To get service: kubectl get service

To describe service: kubectl describe service <service_name>

To get a detailed description of pods: kubectl get pod -o wide

To get the status of deployment: kubectl get deployment <deployment_name> -o yaml > <deployment_name>-result.yaml

---

Demo Project: Complete Application Setup with Kubernetes Components

We will deploy two apps: mongodb and mongo express

Steps:

- Create a mongodb pod and communicate with the pod using an internal service
- Create a mongo express deployment
- Deployment Configuration file to pass DB URL (ConfigMap) and DB credentials (Secret) through environmental variables.
- Create an external service to MongoExpress to be accessible via browser.

1. MongoDB Deployment & Secret

- Create a mongo deployment file with required configurations.
- Create Environment Variables to pass DB information.
- Create a secret.yaml file to store DB credentials to pass to mongo-deployment.
- Encode credentials to base4 before storing them to secret.yaml .
- Apply secret before creating mongo deployment

    kubectl apply -f mongo-secret.yaml
- Apply mongo deployment

    kubectl apply -f mongo.yaml
- Create an internal service within the mongo deployment file to communicate to the Pod

2. MongoExpress Deployment & ConfigMap

- Create MongoExpress deployment with required configuration.
- Assign Environment variables that reference to Secret (Credentials) and ConfigMap (DB Url).
- Create a configmap.yaml to carry the DB Url.
- Apply Configmap.yaml then apply MongoExpress.yaml

    kubectl apply -f mongo-configmap.yaml

    kubectl apply -f mongo-express.yaml

- Check the logs of MongoExpress

kubectl logs <pod_name> (We can get pod name by using "kubectl get pod")

- Create an external service, it is similar to the internal service but what differs is the type defined (type: LoadBalancer) and nodePort(Port for external IP address)
- Once external service is created, to get the external IP

minikube service <service_name> (for us it is "mongo-express-service")

- Now we have a cluster up and running that can be accessed from the browser

Workflow:

Browser -> External Service -> mongoExpress -> Internal Service -> mongodb

---

Organizing components with Kubernetes Namespaces

What is a **NameSpace**?

In Kubernetes cluster we can organize resources in namespaces, so we can have multiple namespaces in a cluster. We can think of namespace as a virtual cluster inside of a kubernetes cluster. While creating a cluster in kubernetes it provides 4 namespaces by default:

kubernetes-dashboard : assigned only for mini-kube installation, not in standard cluster.

-> kube-system: Has system processes, master and kubectl processes.

-> kube-public: Has publicly accessible data. Has ConfigMap(contains cluster info).

-> kube-node-lease: Holds information about heart beats of nodes. Each node has associated lease object in namespace that contains information about the node's availability.

-> default: Used to create resources

How to create a namespace

-> Using "kubectl create namespace <name>"

-> Using a configuration file

Namespaces are used to group resources the reason is that grouping resources helps better identify and access the required resources easily and helps organize the resources in a cluster.

Creating a component in a NameSpace

Using kubectl command

kubectl apply -f <configuration_file_name> --namespace=<namespace_name>

Using Configuration file

Under meta-data add a field namespace and assign a name(namespace: <namespace_name>)

Change a Active NameSpace:

This is helpful change the default NameSpace from Default to a preferred NameSpace. There is no solution for this provided by kubectl so in order to achieve it we need a tool called kubens(can be installed by installing kubectx).

Kubernetes Ingress Explained

Even though we can access an application via an External Service it will contain the web-address having the IP and port number and a service is open for external requests as well. But with ingress we can receive external requests and send them to an internal service while providing a proper domain name to access the application.

Ingress Configuration file:

The Configuration file has *kind: Ingress* which defines the type we need as Ingress and in the Specification we have rules(Routing Rules) this defines the main address requests to that host must be forwarded to an Internal Service.

Configuring Ingress in a Cluster:

Creating just an Ingress component is not enough for routing rules to work, it requires an implementation for Ingress called Ingress Controller(we need to install an IngressController which is basically a Pod/set of Pods that run on the Node in kubernetes cluster) this does evaluation and processing of Ingress rules.

IngressController - Evaluates the rules, Manages Redirections, Entrypoint to cluster.

IngressController using minikube:

To install IngressController to minikube:

minikube add-ons enable ingress

This automatically starts the Kubernetes Nginx Implementation of Ingress Controller.

To see the IngressController:

kubectl get pod -n kube-system

To set rules for IngressController:

Get all the components in Kubernetes dashboard:

kubectl get all -n kubernetes-dashboard

This helps get service name and service ports for defining in IngressController.

Apply the IngressController:

kubectl apply -f dashboard-ingress.yaml

To check the IngressController:

kubectl get ingress -n kubernetes-dashboard

kubectl get ingress -n kubernetes-dashboard --watch

Copy the address generated from the previous command

In Mac: Move to /etc/hosts

Move to the end and paste the ip then space and type"dashboard.com" (mapping ip address)

HELM Package

What is Helm?

It is a package manager for Kubernetes (to package yaml files and distribute them in public and private repositories). Helm Charts are a group of pre-existing YAML files for some common deployments that can be directly downloaded and consumed or can be created and pushed to a repository to be widely available for others to use.

It is used as a templating engine. Template for multiple yaml configurations.

When we want to deploy same application across different environments.

Helm Chart structure:

Directory Structure:

mychart/

Chart.yaml	(has meta information about the chart. eg: name, version, ...)
values.yaml	(place where all the values are configured for template files)
charts/	(will have chart dependencies inside)
templates/	(place where template files are stored)
...	

To install helm: helm install <chart_name>

Kubernetes Volumes

Three components of Kubernetes Storage:

Persistent Volume

Persistent Volume Claim

Storage Class

Persistent volumes are not namespaced and are made available to the whole cluster, for data persistence. Persistent Volume Claims defines the storage needed for the application deployed and the access mode(read,write,..). The PVC is defined in a Yaml file and is linked to the Pod that needs it. The PVC will go to cluster and tries to find a volume that satisfies the claim. The claims must exist in the same namespace as the Pod that uses it.

Kubernetes StatefulSet

What is it?

It is a Kubernetes component that is used specifically for stateful applications. Stateful applications are all databases, any application that stores data to keep track of it's state. Stateless applications do not keep record of their states, each interaction is considered a new one.

Deployment of Stateless & Stateful Applications:

Stateless apps are deployed using deployment component; deployment is an abstraction of pods and allows us to replicate that application in the same cluster. Stateful apps are deployed using StatefulSet components and like deployment, StatefulSet makes it possible to replicate the Stateful app pods and to run multiple replicas of it. In other words they both manage pods that are based on an identical container specification; storage can be configured with both equally in the same way

Deployment vs. StatefulSet:

Replicating Stateful applications is more difficult and has a couple of requirements that stateless applications do not posses. E.g.: Sticky Node Identity, Master Slave nodes.

Kubernetes Services

What is a service and why do we need it?

Service provides a static IP address for accessing Pods.
It provides Load Balancing between Pod and its replica.
It is good for Loose Coupling for communication within and outside the cluster.

Types of Service Attributes:

ClusterIP, NodePort, LoadBalancer.

Types of Services:

ClusterIP Service:

It is the default type of a service. It is used for defining Internal services.

Headless Service:

When a client wants to communicate directly with a specific Pod. In ClusterIP ther service selects a random Pod amidst the replicas to talk to. Eg: Stateful apps(Pod replicas aren't identical). For a client to communicate to a Pod specifically, it needs to find out the IP addresses of each Pod, this can be achieved by:

1. API call to Kubernetes API Server
2. DNS LookUp (Returns a single IP address (ClusterIP))

DNS lookup can be achieved by setting ClusterIP: None in the service. This returns the Pod IP address rather than the Cluster IP address.

NodePort Service:

Creates service accessible on a static port on each worker node on the cluster.
Makes external traffic be able to access static fixed port.
In other words it is an External Service.(NodePort value range: 30000 - 32767).

LoadBalancer Service:

Service becomes accessible externally through a cloud provider's load balancer functionality. This is similar to NodePort but the static Port created is accessible only via the LoadBalancer and not directly.

DOCKER Container

What is a Container?

It is a way to package applications with all necessary dependencies and configuration.
It is a portable artifact easily shared and moved around.
Using containers makes development and deployment process quiet efficient.

Where do containers live?

They live in a Container Repository (a special type of storage for containers), many companies have very own private repositories to host and store containers. There are also public repositories (Docker and DockerHub) for docker container where we can find and use the containers we want.

Basic Docker Commands

docker pull -> Used to pull latest version of an application to the system.
docker run -> Runs the pulled image in a docker container.
docker run --options -> provides various modes to run a container image.
docker stop -> takes the container id that needs to be stopped.
docker start -> takes the container id that needs to be started.
docker ps -> Provides a list of running docker containers.
docker exec -it -> takes a container id and the preferred CLI to open a CLI for the given container.
docker logs -> takes a container id to display the logs of the particular container.

Docker in Practice

E.g.: Working on a simple JS app with mongoDB and mongoDB Express containers.

Step 1: Create a JS app.
Step 2: Pull mongoDB and mongoDB Express containers from DockerHub.
Step 3: Connect both the containers (using Docker Network).
Step 4: Docker provides certain network by default, to check run "docker network ls".
Step 5: Create a docker network, to do so run:"docker network create <network_name>"
Step 6: Run the docker container for mongoDB, run "docker run -p 27017:27017 -d <environment_variables> --net <network_name> mongo"(has mongo ports, environment variables can be viewed from dockerhub-mongo)

Docker SetUp

Docker Prerequisites:

Virtualization needs to be enabled.
Requires Microsoft Hyper-V to run. If not running Docker installer will enable it.
Once Hyper-V is enabled, VirtualBox will no longer work.
Works natively on Windows 10 (64bit).
Docker for Windows include: Docker Engine, Docker CLI client, Docker Compose, Docker Machine & KiteMatic.

Installation for Windows:

Check Prerequisites.
Download "Get Docker for Windows[Stable]".
Once downloaded click on the installer and complete installation.
Docker for Windows.
Start Docker manually by searching Docker app and click on it.
