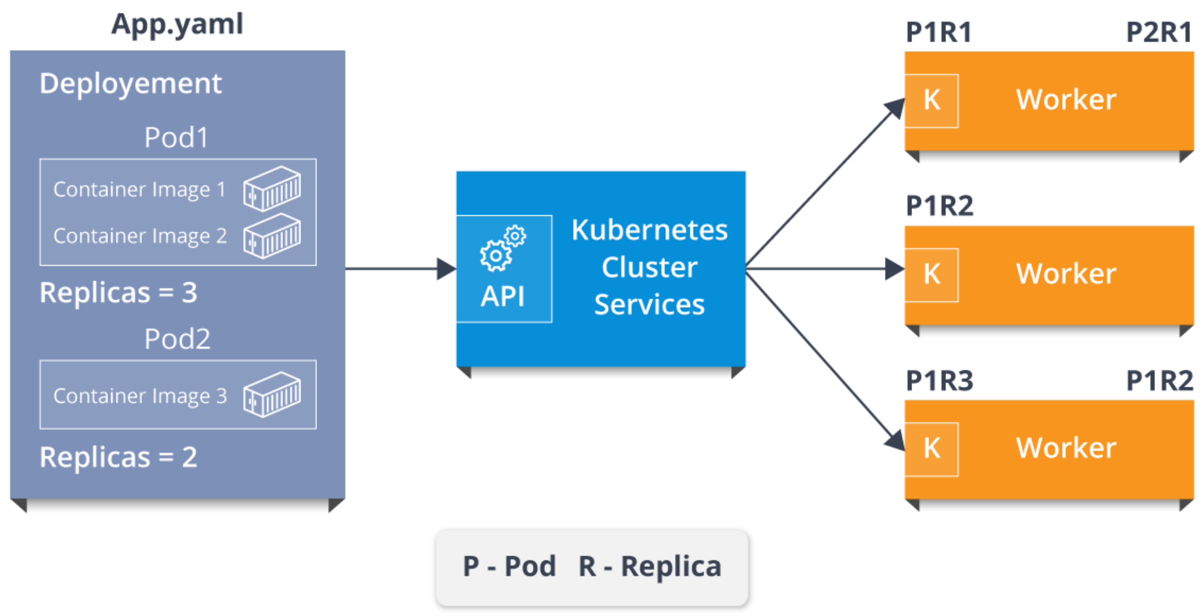
Now, anybody working with Kubernetes must have a clear understanding of a Kubernetes cluster as that will help you in understanding Kubernetes networking.

The Kubernetes platform offers desired state management, which enables the cluster services to run, the fed configuration in the infrastructure. Let me explain with an example.

Consider a YAML file which has all the configuration information which needs to be fed into the cluster services. This file is fed to the API of cluster services, and then it will be up to the cluster services to figure out how to schedule pods in the environment. So, suppose there are two container images for pod 1 with three replicas, and one container image for pod 2 with two replicas. It will be up to the cluster services to allocate these pod-replica pairs to the workers.



Refer to the above diagram. The cluster services have allotted the first worker with two pod replica pairs, the second worker with a single pod-replica pair, and the third worker with two pod replica pairs. Now, it is the Kubelet process which is responsible for communicating the cluster services with workers.

So, this whole setup of cluster services and the workers themselves makes up this Kubernetes cluster!

How do you think these individually allocated pods communicate with each other?

The answer is Kubernetes networking!

There are mainly 4 problems to solve with the networking concepts.

- Container-to-container communication
- Pod-to-pod communication
- Pod-to-service communication
- External-to-service Communication

Now, let me tell you how are the above problems are solved with Kubernetes networking.

Kubernetes Networking

The communication between pods, services, and external services to the ones in a cluster brings in the concept of Kubernetes networking.

So, for your better understanding let me divide the concepts into the following.

- Pods and Container Communication
- Connecting External to Services via Ingress Network

Pods and Container Communication

Pods are basic units of Kubernetes applications that consist of one or more containers allocated on the same host to share a network stack and other resources. This implies that all containers in a pod can reach other on a local host.

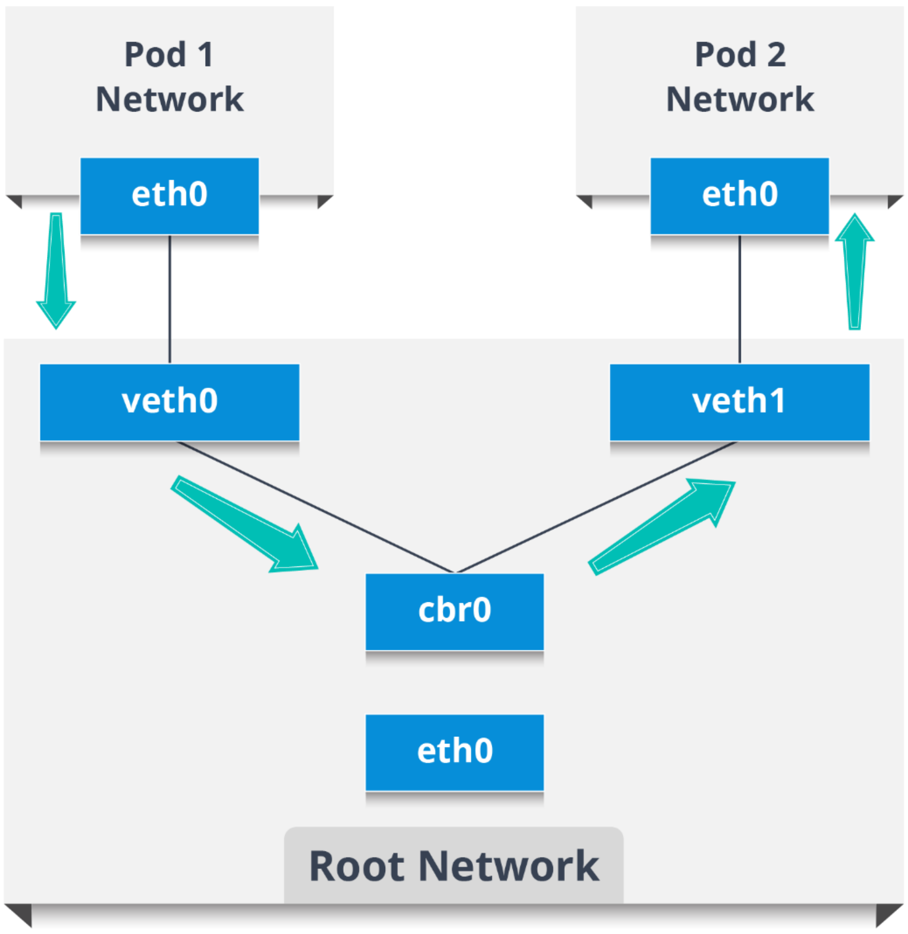
There are 2 types of communication, inter-node communication intra-node communication.

So, let's start with intra-node communication, but before that let me introduce to you the components of the pod network.

Intra-node Pod Network

An intra-node pod network is basically the communication between two different nodes on the same pod. Here's an example.

Assume a packet is going from pod 1 to pod 2.

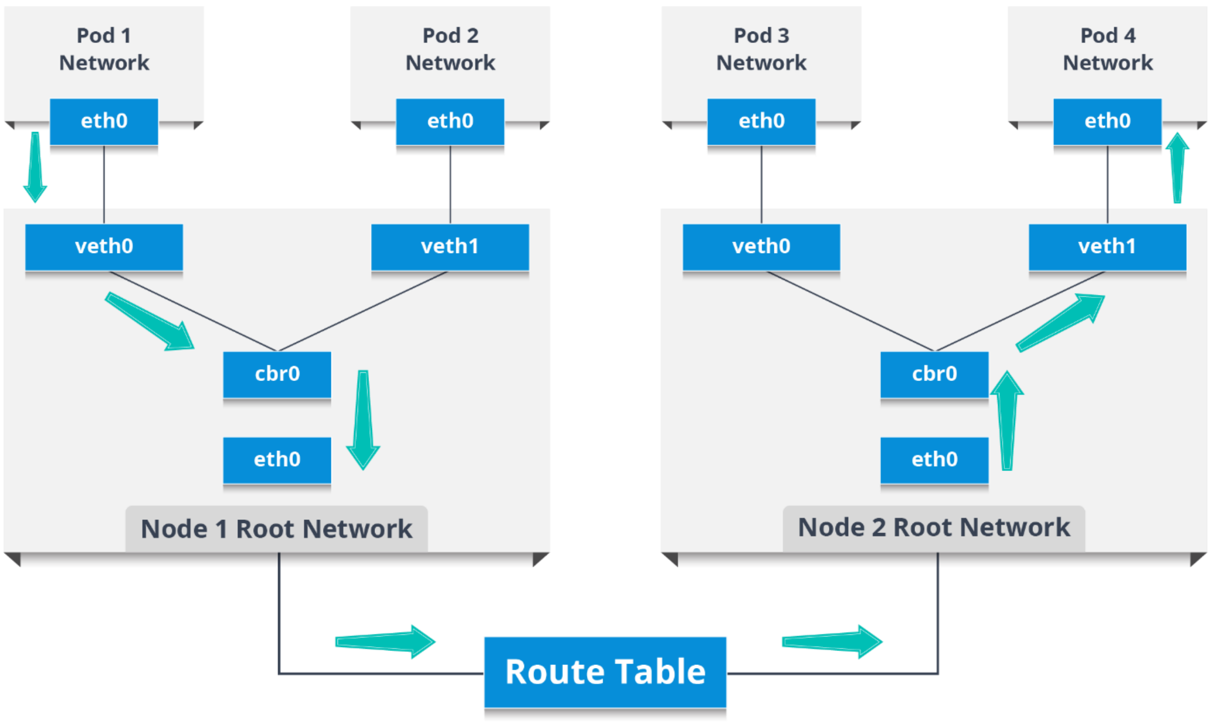


- The packet leaves Pod 1's network at eth0 and enters the root network at veth0
- Then, the packet passes onto the Linux bridge (cbr0) which discovers the destination using an ARP request.
- So, if veth1 has the IP, the bridge now knows where to forward the packet.

Now, similarly let me tell you about the inter-node pod communication.

Inter-node Pod Network

Consider two nodes with various network namespaces, network interfaces, and a Linux bridge.



Now, assume a packet travels from pod 1 to a pod 4, which is on a different node.

- The packet leaves the pod 1 network and enters the root network at veth0.
- Then the packet passes on to the Linux bridge (cbr0) whose responsibility is to make an ARP request to find the destination.
- After the bridge realizes that this pod doesn't have the destination address, the packet comes back to the main network interface eth0.
- The packet now leaves the node 1 to find it's destination on the other node and enters the route table which routes the packet to the node whose CIDR block contains the pod 4.
- Now the packet reaches node 2 and then the bridge takes the packet which makes an ARP request to find out that the IP belonging to veth0.
- Finally, the packet crosses the pipe-pair and reaches pod 4.

So, that's how pods communicate with each other. Now, let's move on and see how services help in the communication of pods.

So, what do you think the services are?

Services

Basically, services are a type of resource that configures a proxy to forward the requests to a set of pods, which will receive traffic and is determined by the selector. Once the service is created, it has an assigned IP address which will accept requests on the port.

Now, there are various service types that give you the option for exposing a service outside of your cluster IP address.

There are mainly 4 types of services.

- **ClusterIP:** This is the default service type which exposes the service on a cluster-internal IP by making the service only reachable within the cluster.
- **NodePort:** This exposes the service on each Node's IP at a static port. Since, a ClusterIP service, to which the NodePort service will route, is automatically created. We can contact the NodePort service outside the cluster.
- **LoadBalancer:** This is the service type which exposes the service externally using a cloud provider's load balancer. The NodePort and ClusterIP services, to which the external load balancer will route, are automatically created.
- **ExternalName**: This service type maps the service to the contents of the externalName field by returning a CNAME record with its value.

Now, you might be wondering how external services connect to these networks.

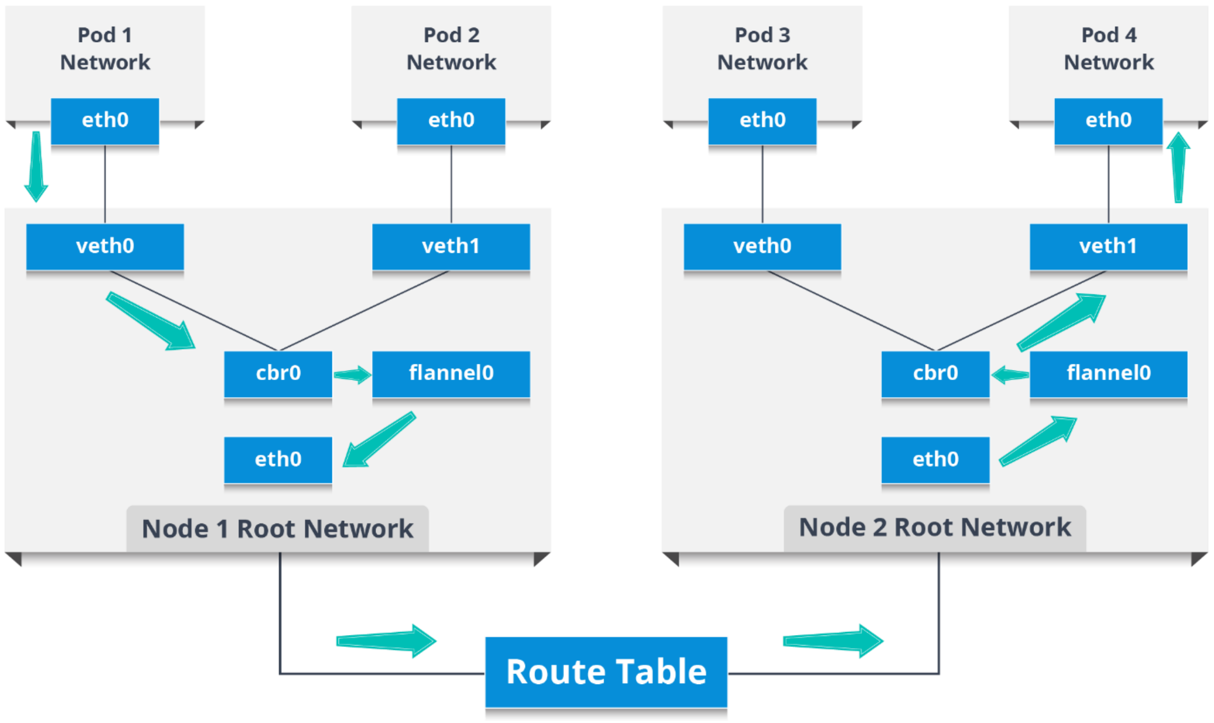
Well, that's by none other than an Ingress network.

An Ingress network is the most powerful way of exposing services as it is a collection of rules that allow inbound connections, that can be configured to give services externally through reachable URLs. So, it basically acts as an entry point to the Kubernetes cluster that manages external access to the services in a cluster.

Now, let me explain to you the workings of Ingress Network with an example.

We have 2 nodes, with the pod and root network namespaces with a Linux bridge. In addition to this, we also have a new virtual ethernet device called flannel0 (network plugin) added to the root network.

Now, we want the packet to flow from pod 1 to pod 4.



- So, the packet leaves pod 1's network at eth0 and enters the root network at veth0.
- Then it is passed on to cbr0, which makes the ARP request to find the destination and it then finds out that nobody on this node has the destination IP address.
- So, the bridge sends the packet to flannel0 as the node's route table is configured with flannel0.
- The flannel daemon talks to the API server of Kubernetes to know all the pod IPs and their respective nodes to create mappings for pods IPs to node IPs.
- The network plugin wraps this packet in a UDP packet with extra headers changing the source and destination IP's to their respective nodes and sends this packet out via eth0.
- Now, since the route table already knows how to route traffic between nodes, it sends the packet to the destination node 2.
- The packet arrives at eth0 of node 2 and goes back to flannel0 to de-capsulate and emits it back in the root network namespace.
- Again, the packet is forwarded to the Linux bridge to make an ARP request to find out the IP that belongs to veth1.
- The packet finally crosses the root network and reaches the destination Pod 4.

So, that's how external services are connected with the help of an ingress network. Now, as I was talking about Network plugins, let me introduce you to the list of popular network plugins available.

Now, that I have told you so much about Kubernetes networking, let me show you a real-life case study.

Case Study: Wealth Wizard Using Kubernetes Networking

Wealth Wizards is an online financial planning platform that combines financial planning, and smart software technology to deliver expert advice at an affordable cost.

It was extremely important for the company to quickly discover and eliminate code vulnerabilities with full visibility of their cloud environment but wanted to control traffic through access restrictions.

So, they used Kubernetes infrastructure to manage the provisioning and rollout of the clusters with the help of tools to manage the deployment and configuration of microservices across the Kube clusters.

They also used a network policy feature of Kubernetes to allow them to control traffic through access restrictions.

Now, the problem was, these policies are application-oriented and can only evolve with the applications, but there was no component to enforce these policies.

The only solution the company could find for this was to use a network plugin, and so they started using Weave Net.

This network plugin creates a virtual network that has a network policy controller to manage and enforce the rules in Kubernetes. Not only this, but it also connects Docker containers across multiple hosts and enables their automatic discovery.

Suppose you have a workload in the cluster and you want to stop any other workload in the cluster talking to it. You can achieve this by creating a network policy that restricts access and only allows ingress to it via the ingress controller on a specific port.

Now, with his deployment on each Kubernetes node, the plugin manages inter-pod routing and has access to manipulate the IPtables rules. In simple terms, each policy is converted to a collection of IPtables rules, coordinated and configured across each machine to translate the Kubernetes tags.

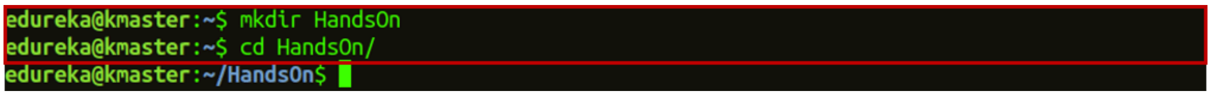
Alright, now that you have gone through so much theory about Kubernetes networking, let me show you how is it done practically.

Suppose you want to store a product name and product ID; for that you will need a web application. Basically, you need one container for web application and you need one more container as MySQL for the backend, and that MySQL container should be linked to the web application container.

Let's get started!

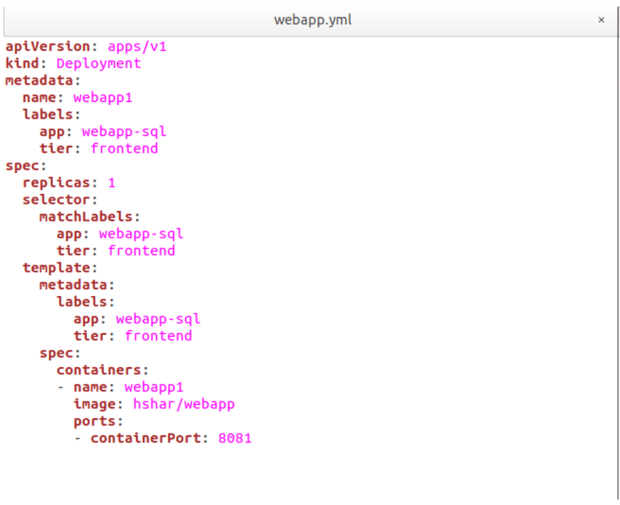
Step 1: Create a folder in your desired directory and change the working directory path to that folder.

```
mkdir HandsOn
cd HandsOn/
```

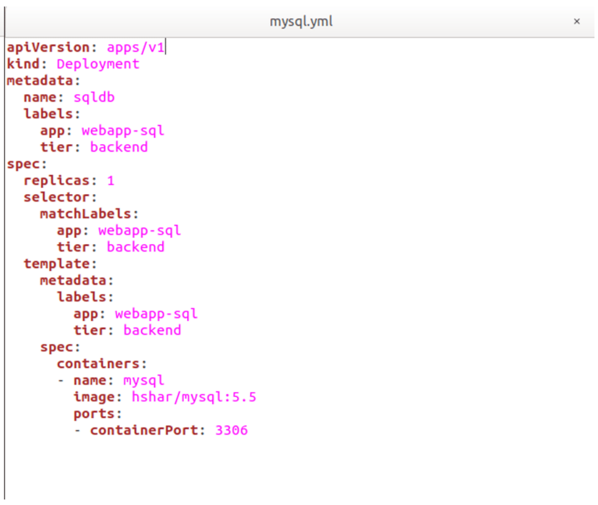


```
edureka@kmaster:~$ mkdir HandsOn
edureka@kmaster:~$ cd HandsOn/
edureka@kmaster:~/HandsOn$
```

Step 2: Now create deployment YAML files, for the web application and MySQL database.



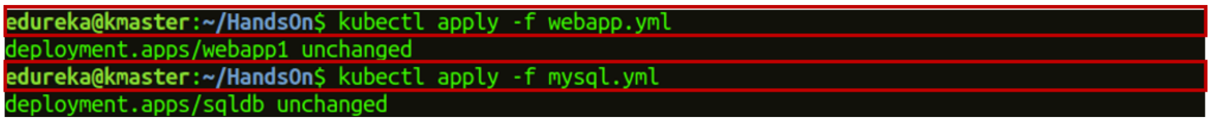
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp1
  labels:
    app: webapp-sql
    tier: frontend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webapp-sql
      tier: frontend
  template:
    metadata:
      labels:
        app: webapp-sql
        tier: frontend
    spec:
      containers:
      - name: webapp1
        image: hshar/webapp
        ports:
        - containerPort: 8081
```



```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sqldb
  labels:
    app: webapp-sql
    tier: backend
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webapp-sql
      tier: backend
  template:
    metadata:
      labels:
        app: webapp-sql
        tier: backend
    spec:
      containers:
      - name: mysql
        image: hshar/mysql:5.5
        ports:
        - containerPort: 3306
```

Step 3: Once you create the deployment files, deploy both the applications.

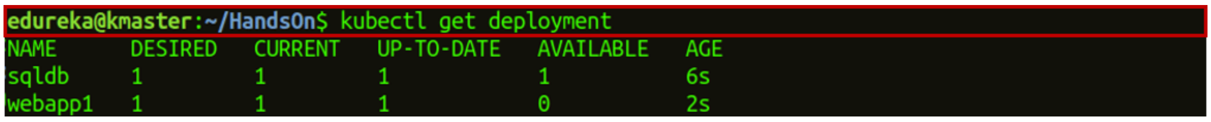
```
kubectl apply -f webapp.yml
kubectl apply -f mysql.yml
```



```
edureka@kmaster:~/HandsOn$ kubectl apply -f webapp.yml
deployment.apps/webapp1 unchanged
edureka@kmaster:~/HandsOn$ kubectl apply -f mysql.yml
deployment.apps/sqldb unchanged
```

Step 3.1: Check both the deployments.

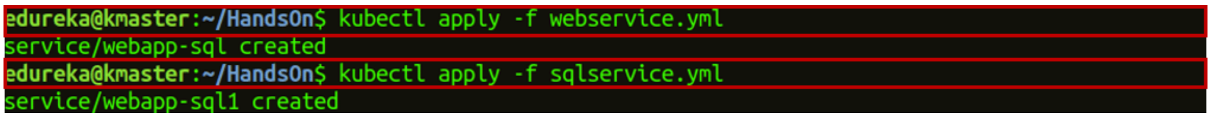
```
kubectl get deployment
```



```
edureka@kmaster:~/HandsOn$ kubectl get deployment
NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
sqldb     1         1         1            1           6s
webapp1   1         1         1            0           2s
```

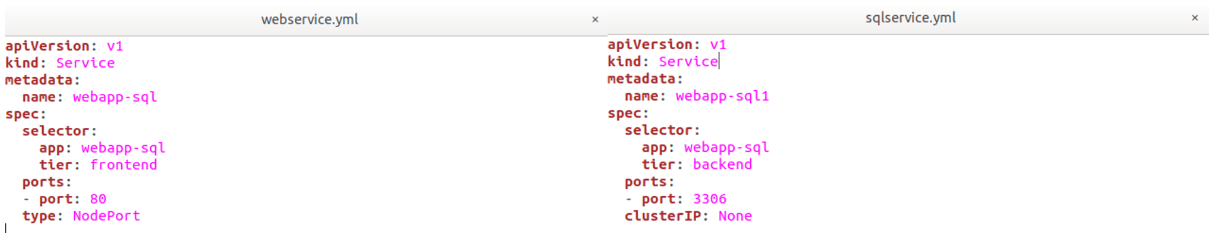
Step 4: Now, you have to create services for both the applications.

```
kubectl apply -f webservice.yml
kubectl apply -f sqlservice.yml
```



```
edureka@kmaster:~/HandsOn$ kubectl apply -f webservice.yml
service/webapp-sql created
edureka@kmaster:~/HandsOn$ kubectl apply -f sqlservice.yml
service/webapp-sql1 created
```

Step 4.1: Once the services are created, deploy the services.



webservice.yml

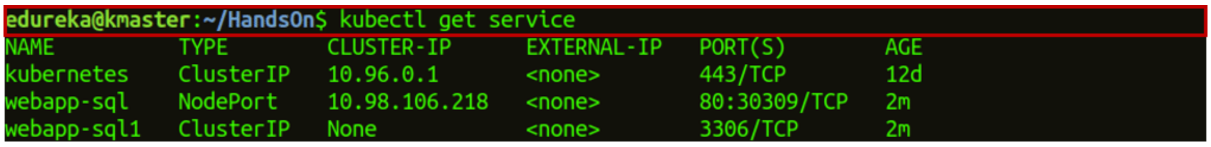
```
apiVersion: v1
kind: Service
metadata:
  name: webapp-sql
spec:
  selector:
    app: webapp-sql
    tier: frontend
  ports:
  - port: 80
  type: NodePort
```

sqlservice.yml

```
apiVersion: v1
kind: Service
metadata:
  name: webapp-sql1
spec:
  selector:
    app: webapp-sql
    tier: backend
  ports:
  - port: 3306
  clusterIP: None
```

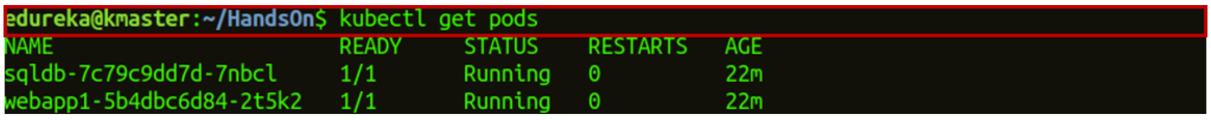
Step 4.2: Check whether the services have been created or not.

```
kubectl get service
```


```
edureka@kmaster:~/HandsOn$ kubectl get service
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP        12d
webapp-sql    NodePort    10.98.106.218   <none>        80:30309/TCP   2m
webapp-sql1   ClusterIP   None            <none>        3306/TCP       2m
```

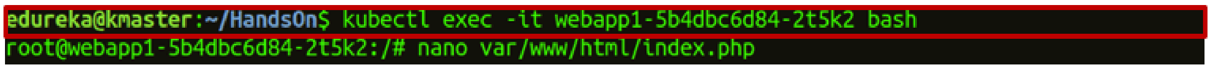
Step 5: Now, check the configuration of running pods.

```
kubectl get pods
```

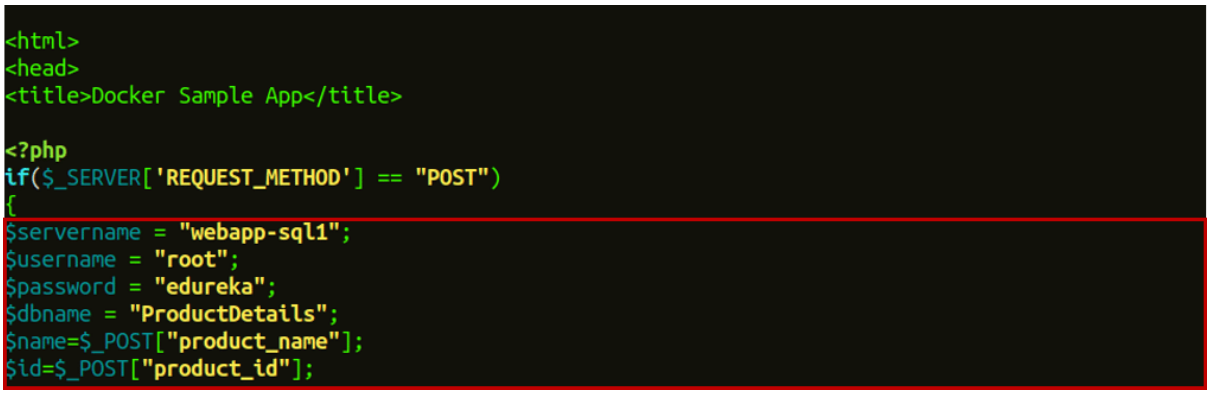

```
edureka@kmaster:~/HandsOn$ kubectl get pods
NAME                          READY     STATUS    RESTARTS   AGE
sqldb-7c79c9dd7d-7nbcl        1/1       Running   0          22m
webapp1-5b4dbc6d84-2t5k2      1/1       Running   0          22m
```

Step 6: Go into the container inside the webapp pod.

```
kubectl exec -it container_id bash
nano var/www/html/index.php
```


```
edureka@kmaster:~/HandsOn$ kubectl exec -it webapp1-5b4dbc6d84-2t5k2 bash
root@webapp1-5b4dbc6d84-2t5k2:/# nano var/www/html/index.php
```

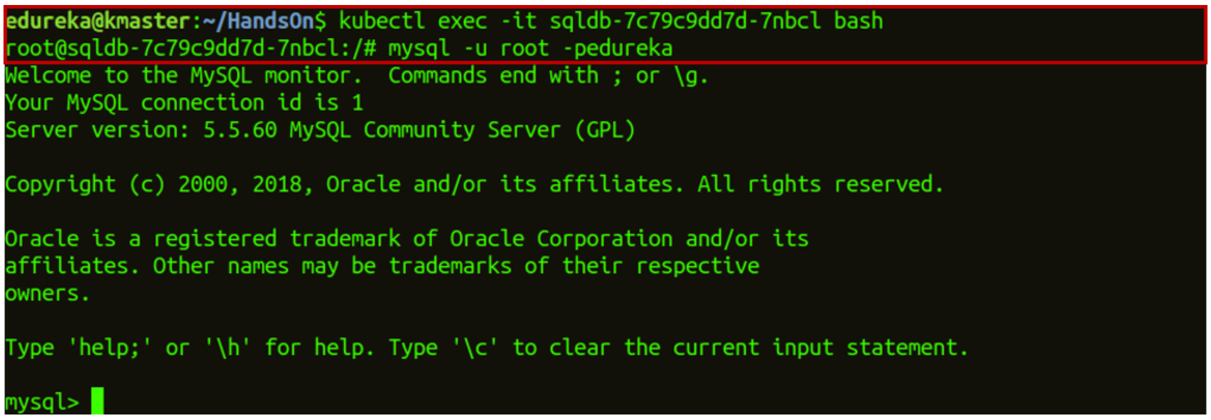
Step 6.1: Now, change the `$servername` from localhost to the SQL service name which is `webapp-sql1` in this case, and `$password` from "" to `edureka`. Also, fill all the database details required and save your index.php file by using the keyboard shortcut Ctrl+x and after that press y to save and press enter.


```
<html>
<head>
<title>Docker Sample App</title>

<?php
if($_SERVER['REQUEST_METHOD'] == "POST")
{
$servername = "webapp-sql1";
$username = "root";
$password = "edureka";
$dbname = "ProductDetails";
$name=$_POST["product_name"];
$id=$_POST["product_id"];
```

Step 7: Now, go into the MySQL container present in the pod.

```
kubectl exec it container_id bash
```


```
edureka@kmaster:~/HandsOn$ kubectl exec -it sqldb-7c79c9dd7d-7nbcl bash
root@sqldb-7c79c9dd7d-7nbcl:/# mysql -u root -pedureka
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 5.5.60 MySQL Community Server (GPL)

Copyright (c) 2000, 2018, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> 
```

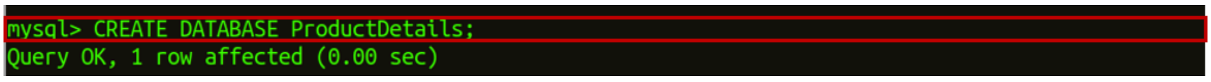
Step 7.1: Get the access to use the MySQL container.

```
mysql -u root -p edureka
```

Where `-u` represents the user and `-p` is the password of your machine.

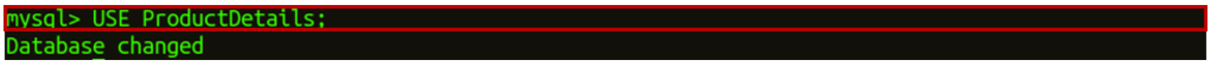
Step 7.2: Create a database in MySQL which will be used to get data from webapp.

```
CREATE DATABASE ProductDetails;
```


```
mysql> CREATE DATABASE ProductDetails;
Query OK, 1 row affected (0.00 sec)
```

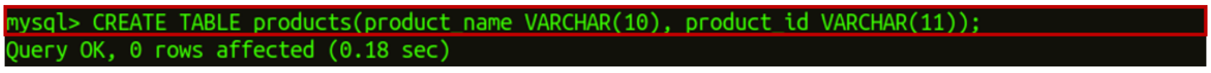
Step 7.3: Use the created database.

```
USE ProductDetails;
```


```
mysql> USE ProductDetails;
Database changed
```

Step 7.4: Create a table in this database in MySQL which will be used to get data from webapp.

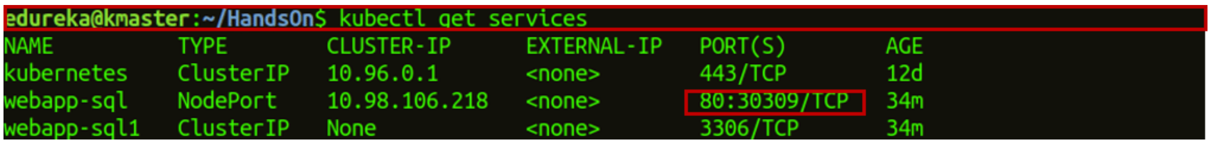
```
CREATE TABLE products(product_name VARCHAR(10), product_id VARCHAR(11));
```


```
mysql> CREATE TABLE products(product_name VARCHAR(10), product_id VARCHAR(11));
Query OK, 0 rows affected (0.18 sec)
```

Step 7.5: Now, exit MySQL container as well using the command exit.

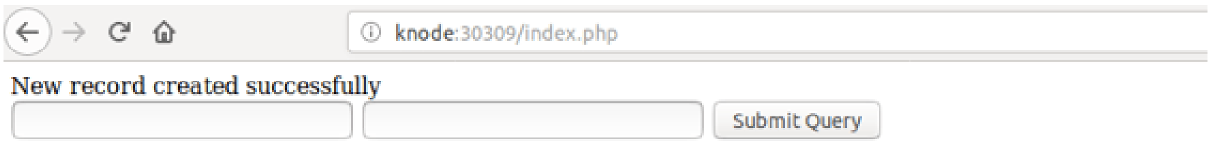
Step 8: Check the port number on which your web application is working.

```
kubectl get services
```



```
edureka@kmaster:~/HandsOn$ kubectl get services
NAME          TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)        AGE
kubernetes    ClusterIP   10.96.0.1       <none>        443/TCP        12d
webapp-sql    NodePort    10.98.106.218   <none>        80:30309/TCP   34m
webapp-sql1   ClusterIP   None            <none>        3306/TCP       34m
```

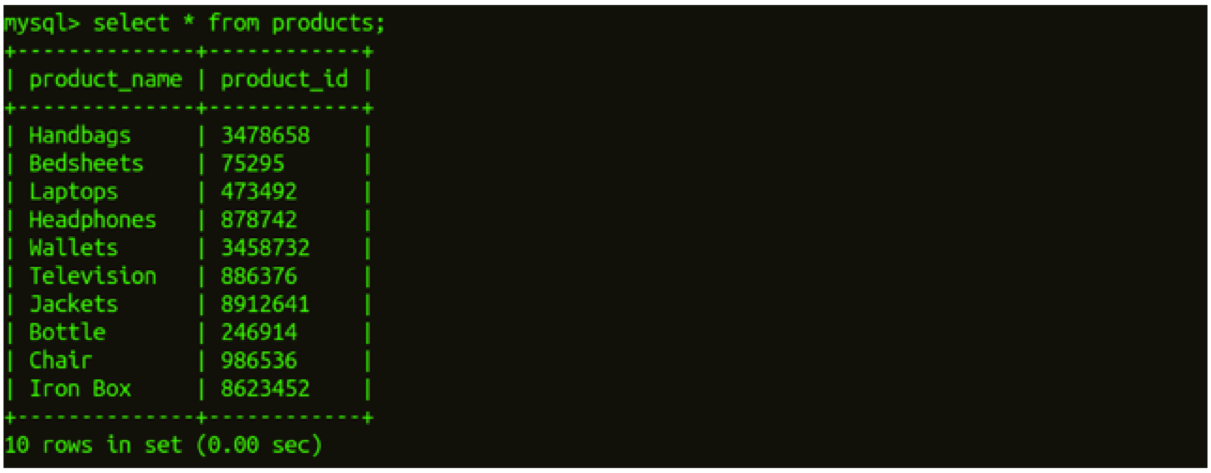
Step 8.1: Now, open the web application on it's allocated port number.



knode:30309/index.php

New record created successfully

Step 9: Once you click on "Submit Query," go to the node in which your MySQL service is running and then go inside the container.



```
mysql> select * from products;
+--------------+------------+
| product_name | product_id |
+--------------+------------+
| Handbags     | 3478658    |
| Bedsheets    | 75295      |
| Laptops      | 473492     |
| Headphones   | 878742     |
| Wallets      | 3458732    |
| Television   | 886376     |
| Jackets      | 8912641    |
| Bottle       | 246914     |
| Chair        | 986536     |
| Iron Box     | 8623452    |
+--------------+------------+
10 rows in set (0.00 sec)
```

This will show you the output of all the list products, of which you have filled in the details.