Summary

Kubernetes Networking

- A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.
- NetworkPolicy resources use labels to select pods and define rules which specify what traffic is allowed to the selected pods.
- By default, pods are non-isolated; they accept traffic from any source.
- Pods become isolated by having a NetworkPolicy that selects them. Once there is any NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed by any NetworkPolicy. (Other pods in the namespace that are not selected by any NetworkPolicy will continue to accept all traffic.)

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

Cluster Networking - Internal

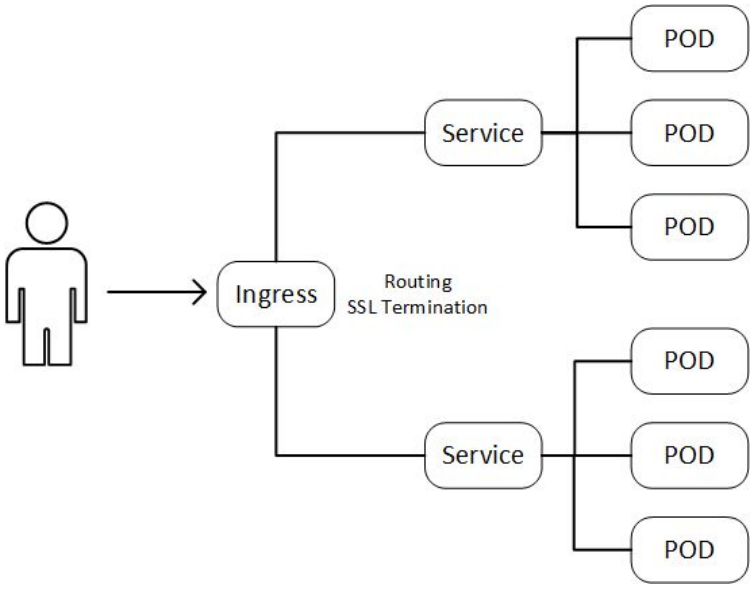
- Every Pod gets its own IP address.
- This means you do not need to explicitly create links between Pods and you almost never need to deal with mapping container ports to host ports.
- This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.
- Kubernetes imposes the following fundamental requirements on any networking implementation (barring any intentional network segmentation policies):
    - pods on a node can communicate with all pods on all nodes without NAT agents on a node (e.g. system daemons, kubelet)
    - can communicate with all pods on that node
    - pods in the host network of a node can communicate with all pods on all nodes without NAT
    - This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers.
    - If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.

Kubernetes DNS

- Every Service defined in the cluster (including the DNS server itself) is assigned a DNS name. By default, a client Pod's DNS search list will include the Pod's own namespace and the cluster's default domain. This is best illustrated by example:
- Assume a Service named foo in the Kubernetes namespace bar. A Pod running in namespace bar can look up this service by simply doing a DNS query for foo. A Pod running in namespace quux can look up this service by doing a DNS query for foo.bar.
- Services are assigned a DNS A record for a name of the form: my-svc.my-namespace.svc.cluster.local
- When enabled, pods are assigned a DNS A record in the form: pod-ip-address.my-namespace.pod.cluster.local
- Pod's DNS Config allows users more control on the DNS settings for a Pod.
  The dnsConfig field is optional and it can work with any dnsPolicy settings. However, when a Pod's dnsPolicy is set to "None", the dnsConfig field has to be specified.

Kubernetes Ingress Controller

- Ingress, added in Kubernetes v1.1, exposes HTTP and HTTPS routes from outside the cluster to services within the cluster.
- Traffic routing is controlled by rules defined on the Ingress resource.
- An Ingress can be configured to give services externally-reachable URLs, load balance traffic, terminate SSL, and offer name based virtual hosting.
- An Ingress controller is responsible for fulfilling the Ingress, usually with a loadbalancer, though it may also configure your edge router or additional frontends to help handle the traffic.
- An Ingress does not expose arbitrary ports or protocols.
- Exposing services other than HTTP and HTTPS to the internet typically uses a service of type Service.Type=NodePort or Service.Type=LoadBalancer.



Kubernetes Volumes & Data

- Managing storage is a distinct problem from managing compute.
- The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed.
- To do this we introduce two new API resources:
    - PersistentVolume
    - PersistentVolumeClaim.
- A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator.
- It is a resource in the cluster just like a node is a cluster resource.
- PVs are volume plugins like Volumes, but have a lifecycle independent of any individual pod that uses the PV.
- PersistentVolume captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.
- A PersistentVolumeClaim (PVC) is a request for storage by a user. It is similar to a pod.
- Pods consume node resources and PVCs consume PV resources.
- Pods can request specific levels of resources (CPU and Memory).
- Claims can request specific size and access modes (e.g., can be mounted once read/write or many times read-only).
- While PersistentVolumeClaims allow a user to consume abstract storage resources, it is common that users need PersistentVolumes with varying properties, such as performance, for different problems.
- Cluster administrators need to be able to offer a variety of PersistentVolumes that differ in more ways than just size and access modes, without exposing users to the details of how those volumes are implemented. For these needs there is the StorageClass resource.
- Persistent Volumes can be provisioned in two ways:
    - Static - A cluster administrator creates a number of PVs. They carry the details of the real storage which is available for use by cluster users. They exist in the Kubernetes API and are available for consumption.
    - Dynamic - When none of the static PVs the administrator created matches a user's PersistentVolumeClaim, the cluster may try to dynamically provision a volume specially for the PVC.

Kubernetes Storage Binding & Reclaim Policy

- A user creates, or has already created in the case of dynamic provisioning, a PersistentVolumeClaim with a specific amount of storage requested and with certain access modes.
- A control loop in the master watches for new PVCs, finds a matching PV (if possible), and binds them together.
- If a PV was dynamically provisioned for a new PVC, the loop will always bind that PV to the PVC.
- Otherwise, the user will always get at least what they asked for, but the volume may be in excess of what was requested.
- Once bound, PersistentVolumeClaim binds are exclusive, regardless of how they were bound. A PVC to PV binding is a one-to-one mapping.
- Claims will remain unbound indefinitely if a matching volume does not exist.

Kubernetes ConfigMaps

- ConfigMaps allow you to decouple configuration artifacts from image content to keep containerized applications portable.
- ConfigMap resource holds key-value pairs of configuration data that can be consumed in pods or used to store configuration data for system components such as controllers.
- ConfigMap designed to more conveniently support working with strings that do not contain sensitive information.
- ConfigMaps are not intended to act as a replacement for a properties file.
- ConfigMaps are intended to act as a reference to multiple properties files.
- You can think of them as way to represent something similar to the /etc directory, and the files within, on a Linux computer. One example of this model is creating Kubernetes Volumes from ConfigMaps, where each data item in the ConfigMap becomes a new file.